Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach

Aiko Yamashita

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo

June 2012

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1237

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika publishing.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

A mi mejor amiga Andrea
(en honor a todas nuestras aventuras...)

Abstract

Code smells are indicators of software design shortcomings that can decrease software maintainability. An advantage of code smells over traditional software measures is that the former are associated with an explicit set of refactoring strategies to improve the existing design. As such, code smell analysis is a promising approach to address both the assessment and the improvement of maintainability. An important challenge in code smell analysis is understanding the interplay between code smells and different aspects of maintenance. Research on code smells conducted in the past decade has emphasized the formalization and automated detection of code smells. Much less research has been conducted to empirically investigate how comprehensive and informative code smells are for the assessment of software maintainability. If we are to use code smells to assess maintainability, we need to understand their potential in explaining and predicting different maintenance outcomes and their usefulness in industrial settings. Relevant questions in using code smells as maintainability indicators include: "What and how much can code smells tell me about the maintainability of a system as a whole?" and "How suitable are code smells in identifying code segments (i.e., files) with low software maintainability?" The main goal of this thesis is to empirically investigate, from different perspectives and in a realistic setting, the strengths and limitations of code smells in supporting software maintainability assessments. The secondary goal is to suggest approaches to address the limitations of code smells and the assessments based on them. Both goals are reflected in our attempt to answer the following research questions:

- Research Question 1: How good are code smells as indicators of system-level maintainability of software, and how well do code-smell-based assessments perform compared with other assessment approaches?
- Research Question 2: How good are code smells in distinguishing source code files that are likely to require more maintenance effort than others?
- Research Question 3: How good are code smells in discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance?

- Research Question 4: How much of the total set of problems that occur during maintenance can be explained by the presence of design problems related to code smells?

- Research Question 5: How well do current code smell definitions correspond with maintainability characteristics deemed critical by software developers?

- Research Question 6: How should code smell analysis be combined with expert judgment to achieve better maintainability assessments?

To answer these questions, we conducted a multiple-case study in which a maintenance project–involving four Java web systems, six software professionals, and two software companies–was observed over several weeks. The four systems had almost identical functionalities, which gave us the opportunity to observe how developers performed the same maintenance tasks on systems with different code designs. We used code smells to discriminate between the systems and to characterize the maintainability of each. Information about different maintenance outcomes (e.g., effort and defects) was collected and used to compare the ability of code smells to explain or predict maintenance outcomes, i.e., to determine how differences in the presence of code smells were related to differences in the maintenance outcomes and to what extent. Qualitative data were collected and analyzed to supplement and triangulate the analyses of the relation between code smells and maintenance outcomes.

A main observation derived from our analyses is that the usefulness of code smells depends on the maintainability perspective involved and the particular operationalization of maintainability. Although results of one analysis may appear contradictory to the results of another analysis, this may just indicate different perspectives and/or operationalizations of maintainability. These perspectives and operationalizations are therefore emphasized as the interpretation contexts of our results. Some of the results, which we consider our main research contributions, are listed below.

From a *system-analysis* perspective of maintainability, this thesis contributes the following findings.

- When maintainability was operationalized through *maintenance effort* and *defects*, the number of code smells present in a system was not a better indicator of maintainability than the simpler measure of system size, measured as lines of code (LOC) of a system. When the systems differed largely in size, the use of code smell density (i.e., the number of code smells/LOC) yielded system maintainability assessments that were inconsistent with those derived from a comparison of the systems' maintenance effort and defects. Code smell density was a better measure of maintainability

than the number of code smells only when comparing the maintainability of systems similar in size. Expert-judgment-based assessment was a more accurate and flexible approach for system-level maintainability assessments than code-smell-based and C&K-metric-based assessment approaches. An advantage of expert-judgment-based assessments was that they were able to include adjustments related to differences in system size and complexity and to consider the effect of different maintenance scenarios. In spite of this advantage of expert judgment, we found that the use of code smells can complement the expert-judgment-based assessment approach because code smells were able to identify critical code that experts overlooked.

- When maintainability was operationalized through measures of the occurrence of problems[1] during maintenance, the role of code smells on the overall system maintainability was relatively small. Of the total set of maintenance problems, only about 30% were related to files containing code smells. The majority of maintenance problems were not directly related to the source code at all, such as lack of adequate technical infrastructure and external services.

- When maintainability was operationalized through a set of system-level characteristics deemed important by software developers (e.g., infrastructure, architecture, and external services), many of these characteristics did not directly correspond to current definitions of code smells. Consequently, many maintainability characteristics require the use of other approaches, such as *expert judgment* and *semantic analysis techniques* to be evaluated. However, some important system-level maintainability characteristics displayed better correspondence with the definitions of code smells. "Design consistency," for example, was considered highly important by software developers and at the same time showed high correspondence with the definition of several code smells, including some for which detection strategies are not yet available.

From a *file-level-analysis* perspective of maintainability, this thesis contributes the following findings:

- When maintainability was operationalized through *maintenance effort*, none of the 12 investigated code smells significantly indicated an *increase* in the maintenance effort of files.

- When maintainability was operationalized through the incidence of *maintenance problems*, a violation of the interface segregation principle (ISP) within a file indicated a significantly higher likelihood of problems with that file during maintenance.

[1] Maintenance problems were identified through direct observation and from statements from developers during interviews.

A methodological contribution of our thesis comprises a report on our experiences, insights and recommendations from using the *concept mapping* technique in a software maintainability assessment context. This method was adopted from social research, and we used it to better incorporate input from expert judgment in the context of selection, analysis, and interpretation of code smells during maintainability assessments. The main conclusion is that despite some limitations, concept mapping is a promising approach for maintainability assessments that need to combine different sources of information, such as code smell analysis and expert judgment.

Preface

The work leading to this thesis was performed at Simula Research Laboratory and the Department of Informatics, University of Oslo, with Ole Hanseth as the main supervisor and Magne Jørgensen as the co-supervisor. Bente Anda, Dag Sjøberg, and Tore Dybå initially supervised the work, and Ole Hanseth and Magne Jørgensen took charge in the latter stage. This work was partly funded by Simula Research Laboratory and the Research Council of Norway through the projects AGILE (Grant No. 179851/I40) and TeamIT (Grant No. 193236/I40).

Acknowledgement

When I decided to cross the Atlantic Ocean to start a new adventure, I didn't know I was going after the "white rabbit". For a while, I was bewildered by how deep the the rabbit hole was. Yet, if I were given a second chance, I wouldn't change a thing. These years and the people I have met are now part of me, and part of these pages. To them, and to those who were with me from the beginning, I owe this work. I would like to thank my mentor and friend Magne Jørgensen, for his help and guidance, and for all our interesting conversations. My sincere gratitude to Ole Hanseth for stepping in when I needed help. To Leon Moonen, for his dedication and support, which led to results of excellent quality. To Steve Counsell for a great collaboration. To Bente Anda, for her guidance in the initial stage of the PhD, where the first "big batch" of the work had to be done. To Hans Christian Benestad and Gunnar Bergersen for their support and insightful discussions. To Erik Arisholm for lending me his expertise, and to Dag Sjøberg for inviting me to come to Norway to start a PhD. I would like to thank Simula Research Laboratory for providing me with such an excellent working environment. To 'Drift' for all the technical support; you guys always came to my rescue when I needed it. My special thanks to Aslak Tveito for his support, especially during the final lap. I have to thank all my friends at Simula, who made my days bright and exciting. To my office-mate, and unconditional friend Shaukat, for giving me the opportunity to be his friend, and for putting up with me all these years. To Raj for being so awesome (there are no other words) and for always looking after all of us. To Hadi for his dry humor, I always enjoyed it. To Dominik for his continuous supply of Swiss chocolate, his undepletable optimism and unique remarks. To Molly for being there when I needed a hug. To Dietmar for our conversations on Spanish literature. To James, Alexander, Amir, Ragnhild, Tao, Stein, Tanja, Kristin and Lionel for memorable moments during my PhD. My thanks to all of my friends in Norway, Sweden and Costa Rica for always cheering me up. To Sigurd, for all his coaching. To my dear boyfriend Kristian, for all his love, patience and care. Finally, I would have never managed any of this without the love and support of my family, who never stopped believing in me: thank you for being the pillars of my life.

List of Publications

The following papers are included in this dissertation.

1. Code Smells as System-Level Indicators of Maintainability: An Empirical Study
A. Yamashita and S. Counsell
Submitted to Journal of Systems and Software, 2012

2. Quantifying the Effect of Code Smells on Maintenance Effort
D.I.K. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå
Submitted to IEEE Transactions on Software Engineering, 2012

3. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data
A. Yamashita
Submitted to Journal of Empirical Software Engineering, 2012

4. To What Extent can Code Smell Detection Predict Maintenance Problems? – An Empirical Study
A. Yamashita and L. Moonen
Submitted to Journal of Information and Software Technology, 2012

5. Do Code Smells Reflect Important Maintainability Aspects?
A. Yamashita and L. Moonen
Accepted for publication at the International Conference of Software Maintenance, 2012

6. Using Concept Mapping for Maintainability Assessments
A. Yamashita, B. Anda, D.I.K. Sjøberg, H.C. Benestad, P.E. Arnstad and L. Moonen
In Proceedings of the Third International Symposium of Empirical Software Engineering and Measurement (ESEM2009), Orlando, Florida, October 15, 2009, IEEE Computer Society, pp. 378–389

Additional Publications

In the process of completing this Ph.D. work, I contributed to an industrial case study by analyzing the maintenance problems of a company that uses agile software development. The following publications are not directly cited in this dissertation but gave input and contributed to the choice of research methods and analysis techniques used in the study.

7. Software Entropy in Agile Product Evolution
G. K. Hanssen, A. Yamashita, R. Conradi, and L. Moonen
Published in Proceedings of the 43rd Hawaiian International Conference on System Sciences, 2010

8. Maintenance and Agile Development: Challenges, Opportunities and Future Directions
G. K. Hanssen, A. Yamashita, R. Conradi, and L. Moonen
Published in Proceedings of the 25th IEEE International Conference on Software Maintenance, 2009

Contents

List of Figures

List of Tables

Part 1: Summary of the Thesis

Chapter 1

Introduction

1.1 Research Problem

A major goal during software maintenance and evolution is to manage an increasingly large and complex code base as new releases or improvements are made to the product. Numerous studies report that the largest portion of effort in software projects is allocated to maintenance [14, 53, 2, 116]. Consequently, ensuring the maintainability of software becomes of fundamental importance for software organizations.

Code smells are indicators of software design shortcomings that can potentially decrease software maintainability. An advantage of code smells over traditional software measures (such as maintainability index or cyclomatic complexity) is that code smells are associated with an explicit set of refactoring strategies. As such, code smell analysis is a promising approach to support both maintainability assessment and improvement. Nevertheless, it is not clear how and to which extent code smells can reflect or describe how maintainable a system is. This makes the interpretation and use of code smells somewhat difficult and hinders the possibility of conducting cost-effective refactoring. Given that refactoring represents a certain level of risk (e.g., introduction of defects) and cost (e.g., time spent by developers modifying the code and cost of regression testing), it is essential to weigh the effort and risks of eliminating versus ignoring the presence of code smells. Furthermore, it is important to understand which maintenance aspects can be addressed by code smells and which should be addressed by other means. Insufficient information on maintenance aspects, such as severity levels and the range of effects of code smells, makes refactoring prioritization a nontrivial task. To support cost-effective refactoring, we need to increase our understanding of how code smells affect maintenance, what kinds of difficulties they cause, and how they can affect productivity in a project.

Research in the past decade has emphasized the formalization and automated detection of code smells, but little has been done to investigate how comprehensive and informative code smells are in assessing maintainability in practical settings. Even less empirical work on code smells includes in vivo studies, which limits the applicability of the results in industrial settings. Existing studies represent many different contexts, but often the descriptions of the context of study are insufficient. The results from these studies are hence difficult to interpret and to apply in real-life settings. Moreover, there is a significant lack of theory development in the field, which may partly be due to the small number of in-depth analyses (e.g., based on interpretive research) where observations in the field can be used to derive new theories or modify existing theories of a given software engineering phenomenon.

If we are to use code smells to assess maintainability, we also need to understand better *when* they are useful (e.g., the contexts in which their "predictive power" is acceptable) in predicting and assessing different maintenance aspects as well as their applicability in practical settings. One may ask the following: What and how much can code smells tell me about the maintainability of a system as a whole? How suitable are code smells in identifying code segments (i.e., files) with low software maintainability? Once such questions are satisfactorily answered, improved code-smell-based maintainability assessment methods can be developed and ultimately implemented within the industry.

1.2 Research Goals and Questions

The main goal of this research is to empirically investigate in a realistic setting how useful code smells are in supporting software maintainability assessments. More specifically, this study attempts to answer the following questions:

- ***Research Question 1:*** How good are code smells as indicators of system-level maintainability of software, and how well do code-smell-based assessments perform compared with other assessment approaches?
- ***Research Question 2:*** How good are code smells in distinguishing source code files that are likely to require more maintenance effort than others?
- ***Research Question 3:*** How good are code smells in discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance?
- ***Research Question 4:*** How much of the total set of problems that occur during maintenance can be explained by the presence of design problems related to code smells?

- ***Research Question 5:*** How well do current code smell definitions correspond with maintainability characteristics deemed critical by software developers?

- ***Research Question 6:*** How should code smell analysis be combined with expert judgment to achieve better maintainability assessments?

1.3 Description of the Research Study

To answer the research questions in Section 1.2, we conducted a case study in which a maintenance project was carried out and observed for seven weeks. It involved four Java web systems, six software professionals, and two software companies. This case study was conducted in a way that resembled, as much as possible, a real-life consultancy project where tasks were solved by individual software developers. The systems under study were developed in 2003 by four Norwegian consultancy companies as part of a study by Anda et al. [7]. The four systems conform to the same requirement specification but have considerable differences in their design and implementation. The systems were deployed in 2003 over Simula Research Laboratories' Content Management System (CMS), but in 2007, due to changes in the CMS, it was not longer possible for the systems to remain operational. This provided a realistic setting for a maintenance project based on a real need for adapting and enhancing the systems. The maintenance project involved three software developers from a company in Czech Republic and three from a company in Poland, who individually performed three maintenance tasks on one system and then repeated the same tasks on a second system. The tasks performed on the systems took three to four weeks per developer. The total duration of the project was seven weeks (four weeks in Czech Republic and three weeks in Poland). Because all four systems displayed nearly identical functionalities, we had the opportunity to observe how developers performed the same maintenance tasks on systems with different designs, enabling more robust and comparable observations across the systems. The design of the study aimed at using code smells to discriminate between system designs and to characterize each. The study was also conducted to compare different maintenance outcomes (e.g., effort and defects) to determine if the observed differences across systems in terms of code smells were related to differences in the maintenance outcomes. In addition to the quantitative data about code smells and the studied maintenance outcomes, qualitative data from interviews and think-aloud sessions were collected. The qualitative data were analyzed to enable a more in-depth investigation of the role code smells play in software maintenance and to increase our understanding of the capabilities and limitations of code smells in supporting the assessment of software maintainability.

1.4 Structure of the Thesis

The remaining part of the thesis is organized as follows:

Summary (Part I): Chapter 2 gives an introduction to code smells and software maintainability, alongside a description of the current state of research in code smells. Chapter 3 describes and motivates the methodology applied in this study. Chapter 4 summarizes the answers to the research questions, discusses the implications of the results, presents the limitations of the thesis work, provides recommendations for improvements, and offers directions for future research. Chapter 5 summarizes the research.

Papers (Part II): This part includes the six papers of the thesis. Brief descriptions of the maintainability perspective involved in the papers, the aim of the papers, and the roles of the authors in the work leading to the papers are presented below, alongside the abstracts for each of the papers.

Paper 1: Code Smells as System-level Indicators of Maintainability: An Empirical Study
A. Yamashita and S. Counsell
Submitted to Journal of Systems and Software, 2012

Paper 1 addresses the *system-level* perspective of software maintainability. The aim of the study is to evaluate the suitability of code smells for conducting system-level maintainability assessments. Bente Anda and Dag Sjøberg provided the idea for this work. I participated in the planning of the study in close collaboration with Bente Anda, and was the main responsible for the execution of the study, and the collection of the entire dataset. I conducted the analysis and the writing of the article. I worked in close collaboration with Steve Counsell during the analysis and writing process of the article.

Abstract: Code smells are manifestations of design flaws that can degrade code maintainability if left to fester. The research in this paper investigates the potential of code smells to reflect system-level indicators of maintainability. We report a study where the strengths and limitations of code smells are evaluated against existing evaluation approaches. We evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment and the Chidamber and Kemerer suite of metrics. The systems were maintained over a period up to 4 weeks. During maintenance, effort (person-hours) and number of defects were measured, to validate the different evaluation approaches. Results suggest that code

smells are strongly influenced by size. An implication is that code smells are likely to yield inaccurate results when comparing the maintainability of systems differing in size. When comparing the evaluation approaches, expert judgment was found as the most accurate and flexible since it considered effects due to a system's size and complexity and could adapt to different maintenance scenarios. We also found that code smells complemented expert-based evaluation, since they can identify critical code that experts can sometimes overlook.

Paper 2: Quantifying the Effect of Code Smells on Maintenance Effort
D.I.K. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå
Submitted to IEEE Transactions on Software Engineering, 2012

Paper 2 addresses the *file-level* perspective of software maintainability, in contrast to the system-level perspective addressed by Paper 1. The aim of the study is to determine the quantifiable effects of twelve code smells on maintenance *effort* at file-level, through multiple regression analysis. Dag Sjøberg provided the idea for this work. I participated in the planning of the study in close collaboration with Bente Anda, and was the main responsible for the execution of the study and the collection of the majority of the dataset. I also conducted the summarization and initial analysis of the data. Dag Sjøberg took the lead in the further analysis and the writing of the article, and Audris Mockus contributed with the data analysis. The other authors contributed in the writing process. I contributed in the writing process, and was responsible of several sections of the paper.

Abstract: **Context:** Code smells are assumed to indicate bad design that leads to less maintainable code. However, this assumption has not been investigated in controlled studies with professional software developers. **Aim:** Our aim was to investigate the relationship between code smells and maintenance effort. **Method:** Six developers were hired to perform three maintenance tasks each on four functionally equivalent Java systems originally implemented by different companies. Each developer spent three to four weeks. In total, they modified 298 Java files in the four systems. An Eclipse IDE plug-in measured the exact amount of time a developer spent maintaining each file. A regression was used to explain the effort using file properties, including the number of smells. **Results:** None of the 12 investigated smells was significantly associated with increased effort after we adjusted for file size and the number of changes; Refused Bequest was significantly associated with decreased effort. File size and the number of changes explained most of the variation in effort. **Conclusion:** The effects of code smells on maintenance effort are limited. In general, to reduce maintenance effort, focusing on code size and the

work practices that limit the number of changes may be more beneficial than refactoring code smells.

Paper 3: Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data
A. Yamashita
Submitted to Journal of Empirical Software Engineering, 2012

Paper 3 addresses, as Paper 2, the *file-level* perspective of software maintainability. Instead of focusing on *effort* as in Paper 2, the aim is to investigate the likelihood of a file being *problematic* during maintenance. I participated in the planning of the study in close collaboration with Bente Anda, and participated in the collection of the entire dataset. I provided the idea for the type of analysis for this paper, and carried out the analysis and writing of the paper. During the analysis and writing process, I received guidance from Magne Jørgensen, and received suggestions from Erik Arisholm.

Abstract: Code smells are indicators of deeper design problems that may cause difficulties in the evolution of a software system. This paper investigates the capability of twelve code smells to reflect actual maintenance problems. Four medium-sized systems with equivalent functionality but dissimilar design were examined for code smells. Three change requests were implemented on the systems by six software developers, each of them working for up to four weeks. During that period, we recorded problems faced by developers and the associated Java files on a daily basis. We developed a binary logistic regression model, with "problematic file" as the dependent variable. Twelve code smells, file size, and churn constituted the independent variables. We found that violation of the Interface Segregation Principle (a.k.a. ISP Violation) displayed the strongest connection with maintenance problems. Analysis of the nature of the problems, as reported by the developers in daily interviews and think-aloud sessions, strengthened our view about the relevance of this code smell. We observed, for example, that severe instances of problems related to change propagation were associated with ISP Violation. Based on our results, we recommend that code with ISP Violation should be considered potentially problematic and be prioritized for refactoring.

Paper 4: To What Extent can Maintenance Problems be Predicted by Code Smell Detection? – An Empirical Study
A. Yamashita and L. Moonen
Submitted to Journal of Information and Software Technology, 2012

Paper 4 addresses the *system-level* perspective of maintainability, but instead of observing the *effort* and *defects* at system level as in Paper 1, it focuses on the incidence of problems during the maintenance work. The aim is to investigate the role of code smells on the overall maintainability of a system by determining the proportion of maintenance problems with a connection to source code, and more specifically, the proportion of source code related maintenance problems connected to files that contain any of the twelve code smells investigated. My contributions in this paper are the same as described for Paper 3, and I worked in close collaboration with Leon Moonen during the analysis and the writing process.

Abstract: **Context:** Code smells are indicators of poor coding and design choices that can cause problems during software maintenance and evolution. **Objective:** This study is aimed at a detailed investigation to which extent problems in maintenance projects can be predicted by the detection of currently known code smells. **Method:** A multiple case study was conducted, in which the problems faced by six developers working on four different Java systems were registered on a daily basis, for a period up to four weeks. Where applicable, the files associated to the problems were registered. Code smells were detected in the pre-maintenance version of the systems, using the tools Borland Together and InCode. In-depth examination of quantitative and qualitative data was conducted to determine if the observed problems could be explained by the detected smells. **Results:** From the total set of problems, roughly 30% percent were related to files containing code smells. In addition, interaction effects were observed amongst code smells, and between code smells and other code characteristics, and these effects led to severe problems during maintenance. Code smell interactions were observed between collocated smells (i.e., in the same file), and between coupled smells (i.e., spread over multiple files that were coupled). **Conclusions:** The role of code smells on the overall system maintainability is relatively minor, thus complementary approaches are needed to achieve more comprehensive assessments of maintainability. Moreover, to improve the *explanatory power* of code smells, interaction effects amongst collocated smells and coupled smells should be taken into account during analysis.

Paper 5: Do Code Smells Reflect Important Maintainability Aspects?
A. Yamashita and L. Moonen
Accepted for publication at the International Conference of Software Maintenance, 2012

Paper 5 aims at determining the *relevance* of current code smell definitions for evaluating

maintainability characteristics deemed as essential by software developers. This study investigates the conceptual relatedness between the current *definitions* of code smells, including those with no automated detection strategies available, and a set of maintainability characteristics deemed as critical by software developers. My contributions in this paper are the same as described for Paper 4, and I worked in close collaboration with Leon Moonen during the analysis and the writing process.

Abstract: Code smells are manifestations of design flaws that can degrade code maintainability. As such, the existence of code smells seems an ideal indicator for maintainability assessments. However, to achieve comprehensive and accurate evaluations based on code smells, we need to know how well they reflect factors affecting maintainability. After identifying which maintainability factors are reflected by code smells and which not, we can use complementary means to assess the factors that are not addressed by smells. This paper reports on an empirical study that investigates the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers. We consider two sources for our analysis: (1) expert-based maintainability assessments of four Java systems before they entered a maintenance project, and (2) observations and interviews with professional developers who maintained these systems during 14 working days and implemented a number of change requests.

Paper 6: Using Concept Mapping for Maintainability Assessments
A. Yamashita, B. Anda, D.I.K. Sjøberg, H.C. Benestad, P.E. Arnstad and L. Moonen
In Proceedings of the 3rd International Symposium of Empirical Software Engineering and Measurement, 2009

Paper 6 describes how concept mapping can be used as a structured approach to combine code smell analysis and expert judgment. I provided the initial idea for the paper, conducted the evaluation, and led the writing process of the paper. The other authors participated in the evaluation and contributed in the writing process of the paper.

Abstract: Many important phenomena within software engineering are difficult to define and measure. A good example is software maintainability, which has been the subject of considerable research and is believed to be a critical determinant of total software costs. Yet, there is no common agreement on how to describe and measure software maintainability in a concrete setting. We propose using concept mapping, a well-grounded method used in social research, to operationalize this concept according to a given goal and perspective. We apply this method to describe four systems that were developed as part of

an industrial multiple-case study. The outcome is a conceptual map that displays an arrangement of maintainability constructs, their interrelations and corresponding measures. Our experience is that concept mapping (1) provides a structured way of combining static code analysis and expert judgment; (2) helps tailoring the choice of measures to a particular system context; and (3) supports the mapping between software measures and aspects of software maintainability. As such, it represents a strong addition to existing frameworks for evaluating quality such as ISO/IEC 9126 and GQM, and tools for static measurement of software code. Overall, we find that concept mapping provides a systematic, structured and repeatable method for developing constructs and measures of the phenomenon of interest, and we deem it useful for defining constructs and measures of other aspects of software engineering, in addition to maintainability.

Chapter 2

Background

This chapter introduces and elaborates on the two core concepts in this dissertation: code smells and software maintainability.

2.1 Definition of Code Smells and Its Relevance on Software Design

Code smells were introduced as indicators of software design shortcomings that can potentially decrease software maintainability. Beck and Fowler provided a set of informal descriptions of 22 code smells and associated them with different refactoring strategies that can be applied to improve software design [40, Ch. 3]. These code smells and the associated refactoring strategies are summarized in Tables 2.1, 2.2, and 2.3. As indicated in Ref. [40], a code smell is a potentially poor design choice that can degrade essential aspects of code quality, such as understandability and changeability.

A basic example of a code smell and a potential refactoring to improve a design is given in Figure 2.1. The code smell *Long Parameter List* may reduce readability (e.g., it could be hard to read too many parameters at a time) and modifiability (e.g., it could demand time-consuming changes if the methods are called often). This code smell may also make developers introduce defects because a developer is more prone to make mistakes when the number of parameters in a method call is high. This code smell can be eliminated by applying the *Introduce Parameter Object* refactoring, which consists in creating a new class containing the parameters of the method and passing the new class as the parameter instead.

Code smells have become an established concept for patterns or aspects of software design that may cause problems in the further development and maintenance of the system [40, 72, 99]. As such, they have been suggested as better indicators of code quality

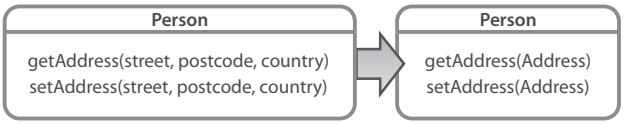

Figure 2.1: Basic example of the elimination of the code smell Long Parameter List by applying the Introduce Parameter Object refactoring strategy.

than traditional object-oriented (OO) metrics. Alshayeb et al. [5] indicated that one of the biggest limitations of traditional OO metrics is their lack of *concrete strategies* (e.g., "knobs") to influence software design positively. From that perspective, code smells have an advantage because each code smell has a specific set of refactoring strategies to improve the design. Consequently, code smells are potentially useful not only for prognosis purposes, but also for improving the quality of the code by associating design problems with concrete solution plans using different refactoring strategies.

Code smells and their refactoring strategies are closely related to OO design principles, heuristics, and patterns. Instances of OO design principles and heuristics can be found in the work by Riel [125] and Coad and Yourdon [24]. Additional work on OO design, which describes concepts such as modularity, information hiding, and open implementation, can be found in Refs. [112, 78, 94, 67]. Work on design patterns (and anti-patterns) can be found in Refs. [19, 42, 73]. Hoss and Carver [57] presented an ontology describing the relationship between anti-patterns and code smells. Martin [90] elaborated on a set of design principles advocated by the Agile community (see Table 2.4).

There has been a growing interest in the topic of code smells for software maintainability assessments since the textbook on code smells by Fowler [40] was published in 1999. Van Emden and Moonen [146] provided the first formalization of code smells and described a tool for Java programs. Mäntylä et al. [84] and Wake [147] proposed two initial taxonomies for code smells. The following subchapters describe the different detection methods developed for code smells and briefly summarize the results of empirical studies on code smells.

Table 2.1: Code smells in [40] and suggested refactoring strategies. (Part 1)

Smells	**Descriptions**	**Suggested Refactoring Strategies**
Alternative Classes with Different Interfaces	Classes that mostly do the same things, but have methods with different signatures	Rename Method Move Method Extract Superclass
Data Class	Classes with fields and getters and setters not implementing any function in particular	Encapsulate Field Encapsulate Collection Remove Setting Method Move Method Extract Method Hide Method
Data Clumps	Clumps of data items that are always found together weather within classes or between classes	Extract Class Introduce Parameter Object Preserve Whole Object
Divergent Change	One class is commonly changed in different ways for different reasons	Extract Class
Duplicated Code	Same or similar code structure repeated within a class or between classes	Extract Method Pull Up Method Form Template Method Substitute Algorithm
Feature Envy	A method that seems more interested in another class other than the one it's actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.	Move Method Extract Method
God (Large) Class	A class has the God Class smell if the class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles.	Extract Class Extract Subclass Extract Interface Duplicate Observed Data
God (Long) Method	A class has the God Method bad smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method	Extract Method Replace Temp with Query Introduce Parameter Object Preserve Whole Object Replace Method with Method Object Decompose Conditional

Table 2.2: Code smells in [40] and suggested refactoring strategies. (Part 2)

Smells	**Descriptions**	**Suggested Refactoring Strategies**
Inappropriate Intimacy	Two classes are overly intertwined	Move Method Move Field Change Bidirectional Association to Unidirectional Extract Class Hide Delegate Replace Inheritance with Delegation
Incomplete library class	Libraries lacking on specific functionality	Introduce Foreign Method Introduce Local Extension
Lazy Class	A class with not enough functionality	Collapse Hierarchy Inline Class
Long Parameter List	Provide methods with just enough data so that it can obtain everything it needs	Replace Parameter with Method Preserve Whole Object Introduce Parameter Object
Message Chains	This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change.	Hide Delegate Extract Method Move Method
Middle Man	A class is delegating almost everything to another class	Remove Middleman Inline Method Replace Delegation with Inheritance
Misplaced Class	In "God Packages" it happens often that a class needs the classes from other packages more than those from its own package.	Move Class
Parallel Inheritance Hierarchies	Each is required to make a subclass of one class, is required also to make a subclass of another	Move Method Move Field

Table 2.3: Code smells in [40] and suggested refactoring strategies. (Part 3)

Smells	**Descriptions**	**Suggested Refactoring Strategies**
Primitive Obsession	Use small objects to represent data such as money (which combines or a date range object)	Replace Data Value with Object Replace Type Code with Class Replace Type Code with Subclasses Replace Type Code with State / Strategy Extract Class Introduce Parameter Object Replace Array with Object
Refused Bequest	Subclasses don't want or need everything they inherit	Push Down Method Push Down Field Replace Inheritance with Delegation
Shotgun Surgery	A change in a class results in the need to make a lot of little changes in several classes	Move Method Move Field Inline Class
Speculative Generality	Over-generalized code in an attempt to predict future needs	Collapse Hierarchy Inline Class Remove Parameter Rename Method
Switch Statements	Conditionals depending of type leading to duplication	Extract Method Move Method Replace Type Code with Subclasses Replace Type Code with State/Strategy Replace Conditional with Polymorphism Replace Parameter with Explicit Methods Introduce Null Object
Temporary Field	Consists of fields that are used as temporary variables. This means that a value assigned to such a field is not used by any method except for the method containing the assignment.	Extract Class Introduce Null Object

Table 2.4: Design principle violations, as described in Martin [90], with suggested patterns.

Design Principle Violations	**Descriptions**	**Suggested Patterns**
Interface Segregation Principle Violation	The dependency of one class to another one should depend on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling	Separation through delegation Separation through multiple inheritance Grouping clients Changing interfaces
Dependency Inversion Principle	High-level modules should not depend upon low-level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions	Polymorphism (abstract classes or interfaces)
Single Responsibility Principle Violation	Each responsibility should be a separate class, because each responsibility is an axis of change. A change to the business rules may cause a class to change, but if SRP is followed, it should not affect other segments of the system (e.g. database schema, GUI, report format, etc)	Façade Pattern Proxy Pattern
Wide Subsystem Interface (Lack of Façade)	A Subsystem Interface consists of classes that are accessible from outside the package they belong to. The flaw refers to the situation where this interface is very wide, which causes a very tight coupling between the package and the rest of the system.	Façade Pattern

2.2 Code Smell Detection and Analysis

The detection of code smells can either be manual or automated. The manual approach normally involves subjective assessment, whereas the automated approaches involve software metrics, heuristics, and tools. Travassos et al. [141] proposed a process based on manual detection to identify code smells for quality assessments. The manual approach poses several limitations, such as lack of scalability and subjective bias. Mäntylä et al. [85] reported that manually detected code smells depend much on the experience level of the subjects performing the detection (e.g., experienced developers detected complex smells more frequently than the less experienced). They also found that developers with less experience with the inspected modules detected more code smells than developers more familiar with the modules.

Most detection approaches for code smells are automated. Earlier work suggested the use of metrics to identify situations where certain refactoring is needed but did not provide any formal specification of code smells. For instance, Simon et al. [132] identified situations when refactoring (i.e., Move Method, Move Attribute, Extract Class, and Inline Class) is potentially needed via identification of use relations among components. Piveta et al. [117] described conditions in which the application of a given refactoring strategy can be advantageous. Yang et al. [151] also proposed a method that identifies areas in need of Extract Method refactoring. Seng et al. [131] proposed an approach for suggesting refactoring based on evolutionary algorithms and simulated refactoring to ensure that a system's externally visible behavior would remain unchanged. Tsantalis and Chatzigeorgiou [145] suggested using volatility as an indicator of areas that need refactoring.

Marinescu [89, 88] suggested a framework for the specification and detection of code smells and provided examples of how to formalize several code smells. Moha et al. [97, 96, 98, 99, 95] proposed a framework to define code smells through a domain-specific language and developed a tool implementing this framework. Alikacem and Sahraoui [4] proposed a language to detect code smells and other violations to code quality principles. Reddy and Rao [123] introduced a set of metrics for describing changes and dependencies between objects over time to detect Shotgun Surgery and Divergent Change smells. Liu et al. [79] proposed a detection and resolution sequence for Duplicated Code, Long Method, Large Class, Long Parameter List, Feature Envy, Primitive Obsession, Useless Field, Method, and Class to simplify their detection and resolution to support more efficient refactoring. Further automated detection methods can be found in Refs. [140, 93, 102, 129, 134, 54, 20].

Tools have also been suggested to visualize and track the evolution of code smells and other metrics. For example, Bakota [10] proposed a tool for tracking the evolution of code

clones, while Mara et al. [86] suggested a domain-specific language for adjusting threshold values used in detection strategies and presented a tool to analyze the evolution of metrics. The work by Marinescu resulted in two commercial tools for code smell detection: Borland Together [17] and InCode [58]. Examples of academic tools for automated code smell detection are JDeodorant, a plug-in to the Eclipse IDE developed by Tsantalis et al. [144] and Fokaefs et al. [38], and the Semantic Web query engine (iSPARQL) developed by Kiefer et al. [68]. Herbold et al. [56] developed a generic framework for the integration of code smell detection techniques and refactoring tools, which was implemented as a plug-in for Eclipse IDE.

Work on code smells has also been extended to cover design properties in aspect-oriented programming (AOP). Monteiro and Fernandes [101] reviewed the traditional OO code smells in the light of aspect orientation and proposed some new code smells for the detection of cross-cutting concerns. Macia et al. [81] proposed a set of detection strategies based on metrics to detect aspect-oriented code smells and related them to important architectural qualities in Ref. [82].

Despite the latest advances on automated code smell detection and several upcoming commercial tools, there are no conclusive answers to whether automated approaches provide better support for refactoring decision than manual approaches. It is also not clear how much additional effort is required to interpret the results from the automated detection of code smells to decide optimally which refactorings must be prioritized over others. For example, Haralambiev et al. [51] asserted that one big challenge of using different software metric tools is their lack of guidance on the interpretation of the metrics. Fontana et al. [39] found that the interpretation of threshold values and measures is challenging across tools. He pointed out that the outcomes of different tools sometimes differ, potentially due to differences in their detection strategies, such as treatment of inner classes, etc. Murphy-Hill [103] presented a set of important qualities for the usefulness of code smell detection tools.

As different detection approaches encompass different features and limitations, some have suggested combining them to achieve better results. For instance, Walter and Pietrzak [148, 115] suggested relying on multiple criteria for code smell detection (e.g., the existence of other smells, expert judgment, and product evolution metrics). Trifu and Reupkes [142] proposed a method that uses a combination of several symptoms (e.g., a combination of code smells) to identify targets for refactoring. Geiger et al. [43] proposed a set of metrics and a visualization technique to identify correlations between a cloned code and change coupling (e.g., “files that are committed at the same time, by the same author, and with the same modification” [43](p.2)) to detect files that need to be restructured.

Van Emden and Moonen [146], Simon et al. [132], Dhambri et al. [32], Parnin et al. [113], Joshi et al. [62], and Carneiro et al. [21] proposed different visualization methods to support either detection, analysis, or interpretation of code smells. Integrated frameworks for analysis and visualization of software characteristics to support semi-automated code inspections can be found in Parnin et al. [113] and Van den Brand et al. [18].

2.3 Empirical Studies on Code Smells

Zhang et al. [155] provided a systematic literature review on code smells and refactoring strategies based on papers published by IEEE and six leading software engineering journals from 2000 to June 2009. The authors found a strong focus on Duplicated Code among the studies investigating code smells, comprising 54% of the identified papers. Nearly half of the identified papers (49%) described methods or tools to detect code smells, one-third (33%) focused on the interpretation of code smells, and 15% centered on refactoring. The review identified only three empirical studies investigating the impact of code smells on maintenance [100, 77, 65]. This review was complemented with a focus on code smells only (i.e., not on refactoring) for the purpose of this thesis, resulting in 11 new papers [31, 30, 69, 80, 63, 66, 107, 28, 108, 121, 1]. The search engines Google Scholar, ACM Digital Library, and ISI Web of Knowledge were used in addition to IEEE Xplore, and the time span was extended to June 2012, using the terms "code smell*," "bad smell*," "bad code smell*," "code-smell*," and "bad-smell*." The following two sections describe the combined results from the empirical studies included in the review by Zhang et al. [155] and those from the extended review.

2.3.1 Studies on the Effects of Code Smells on Maintenance Outcomes

Results on the effects of individual code smells on maintenance outcomes have been reported, related to *defects* [100, 77, 65, 63, 28, 121], *maintenance effort* [31, 30, 80, 1], and *number* and/or *size of changes* [69, 66, 107, 108]. The results are summarized according to the type of code smell provided in Tables 2.5 and 2.6 and discussed briefly below.

Effects on defects: D'Ambros et al. [28] analyzed the code in seven open-source systems (OSSs) and found that neither the presence of Feature Envy nor the Shotg un Surgery code smell was significantly correlated with defects across systems. Juergens et al. [63] observed the proportion of inconsistently maintained duplicated code in relation to the total set of duplicates in C#, Java, and COBOL systems and found (with the

exception of the COBOL system) that 18% of the inconsistent duplicated code was positively associated with defects. Kapser et al. [65] reported on a study where academic experts judged whether they considered duplicated code harmful or not in code pieces of two OSSs. They concluded that some instances of duplicated code were connected with a code that was considered to represent a good style of programming and contributed to a system‘s stability. Consequently, the authors suggested that not all instances of duplicated code should lead to refactoring. Li et al. [77] investigated the relationship between six code smells and defects within classes in an industrial-strength system and found that the presence of the Shotgun Surgery code smell was positively associated with the number of software defects. Monden et al. [100] performed an analysis of a COBOL legacy system and concluded that cloned (duplicated) modules were more reliable but required more effort than noncloned modules. Rahman et al. [121] found that a cloned code is less prone to be defective.

Effects on effort: Abbes et al. [1] conducted an experiment in which 24 students and professionals were asked questions about the code in six OSSs. They concluded that classes and methods identified as God Classes and God Methods in isolation had no effect on effort or quality of responses, but when appearing together, they led to a statistically significant increase in response effort and a statistically significant decrease in the percentage of correct answers, i.e., significantly higher problems with the understanding of the code. Deligiannis et al. [31] conducted an observational study where four participants evaluated two systems, one compliant and one noncompliant with the principle of avoiding the presence of God Classes. Their main conclusion was that the presence of this God Class negatively affected the understandability and the correctness of the answers given by the participants. The same authors also conducted a controlled experiment [30] with 22 undergraduate students as participants. This experiment supported their initial findings that a design without the presence of God Classes resulted in better completeness, correctness, and consistency than a design with this code smell. Lozano et al. [80] compared the maintenance effort of methods in the periods when they contained and did not contain a clone. They found that there was no increase in the maintenance effort in 50% of the periods after methods transitioned from not containing to containing a clone. However, when there was an increase in effort, this increase could be substantial. They report that the effect of clones on maintenance effort depended more on the areas of the system where the clones were located than on the cloning itself.

Effects on changes: Khomh et al. [66] analyzed the source code of Eclipse IDE and found that, in general, classes containing the Data Class code smell were changed more often than classes without it. Kim et al. [69] reported on the analysis of two medium-sized

open-source libraries (Carol and dnsjava) and concluded that 36% of the total amount of code that had been duplicated changed consistently (i.e., they remained as identical duplicates via simultaneous updates), while the remaining evolved independently. Olbrich et al. [107] reported an analysis of several OSSs and found that classes detected to be God Classes and classes with Shotgun Surgery were changed more frequently. However, the same authors found later that classes identified as God Class or Brain Class were changed less frequently and had fewer defects than other classes when class size was used as a covariate [108].

Table 2.5: Studies on the effects of code smells on maintainability (Part 1)

Code Smell	Study	Findings	Method
Duplicated Code	1.Monden et al., 2002 [100]	The modules with duplicated code were more reliable but less maintainable than the modules without such code.	Descriptive analysis of one COBOL legacy system with 2000 modules.
	2.Kim et al., 2005 [69]	36 percent of the duplicated code needed to be changed consistently; the remainder of the duplicated code did not need to be changed in the same direction.	Descriptive analysis of the two medium-sized Java OSSs Carol and dnsjava.
	3.Lozano et al., 2008 [80]	At least 50 percent of the methods with duplicated code required more change effort (partly significant) than the methods without such code.	Nonparametric hypothesis testing of the OSSs GanttProj, jEdit, Freecol, Jboss.
	4.Kapser et al., 2008 [65]	Some of the duplicated code was considered beneficial. Consequently, the authors suggest that not all duplicated code requires refactoring.	Academic experts judged whether Duplicated Code was harmful in pieces of the OSSs Apache and Gnumeric.
	5.Jürgens et al., 2009 [63]	In the Java and C# code, the inconsistently changed duplicated code contained more faults than average code. In the COBOL code, inconsistent changes did not lead to more faults.	Descriptive analysis of 3 industrial C# systems, 1 OSS Java system and 1 industrial COBOL system.
	6.Rahman et al., 2010 [121]	Most of the defective code was not significantly associated with duplicated code. The code that was duplicated less frequently across the system was more error-prone than the code that was duplicated more frequently.	Descriptive analysis and nonparametric hypothesis testing of code and bug tracker in the OSSs Apache httpd, Nautilus, Evolution and Gimp.

2.3.2 Studies on Code Smell Dynamics

Several empirical studies investigated the evolution, longevity, and types of code smells that have a higher tendency to be refactored. Aversano et al. [9] reported that most of

Table 2.6: Studies on the effects of code smells on maintainability (Part 2)

Code Smell	Study	Findings	Method
God Class	7.Deligiannis et al., 2003 [31]	A design (not code) without a God Class was judged and measured to be better (in terms of time and quality) than a design for the same system with a God Class.	Observational case study with four academics as participants.
	8.Deligiannis et al., 2004 [30]	A design (same the design in Study 7) without a God Class had better completeness, correctness and consistency than a design with a God Class.	Controlled experiment over 1.5 hours with 22 undergraduate students as participants.
	9.Olbrich et al., 2009 [107]	The God Classes and classes with Shotgun Surgery were changed more frequently (indicating more maintenance effort) than the other classes. The God Classes had larger churn size, whereas the Shotgun Surgery classes had smaller churn size.	Post-development analysis of the OSSs Lucene and Xerces.
	10.Olbrich et al., 2010 [108]	The God Classes and Brain Classes were changed less frequently and had fewer defects (indicating less maintenance effort) than the other classes.	Nonparametric hypothesis testing of the code and bug-tracker information in the OSSs Lucene, Xerces and Log4j.
God Method	11.Abbes et al., 2011 [1]	The God Classes and God Methods alone had no effect, but compared with the code without both of these smells, the code with the combination of God Class and God Method had a statistically significant increase in effort and a statistically significant decrease in the percentage of correct answers.	Experiment in which 24 students and professionals were asked questions about the code in the OSSs YAMM, JVerFileSystem, AURA, GanttProject, JFreeChart and Xerces.
Data Class	12.Khomh et al., 2010 [66]	Data classes were changed more often than other classes.	Post development analysis in Eclipse. 13 releases were analyzed and looked for correlations on upper 75 of releases.
	13.Li et al., 2007 [77]	Data Class was not associated significantly with software defects.	Analysis of the code and bug-tracker information in the OSS Eclipse.
Refused Bequest		Refused Bequest was not associated significantly with software faults.	
Shotgun Surgery		Shotgun Surgery was positively associated with software faults.	
Feature Envy		Feature Envy was not associated significantly with software defects.	
	14.D'Ambros et al., 2010 [28]	Neither Feature Envy nor Shotgun Surgery was consistently correlated with defects across systems.	Nonparametric hypothesis testing of the code in the OSSs Lucene, Maven, Mina, CDT, PDE, UI, Equinox.

the cloned classes evolved consistently during the same or near in time to a *modification transaction* (i.e., within time window of 400 s). They also found that when the clones did not evolve consistently, this was due to a split in their functionality, indicating that they would follow different evolution paths.

Chatzigeorgiou and Manakos [22] investigated the evolution of a code that contained the Long Method, Feature Envy, and State Checking code smells and found that in most cases, the code smells persisted up to the latest examined version and that the number of code smells accumulated as the system grew. They found that few smells were removed by targeted refactoring work and that the removal was more frequently due to the side effects of adaptive or corrective tasks.

Peters and Zaidman [114] found that code smells have long life spans and were persistent across 50% of the examined revisions. They also found that the code that constituted Feature Envy Methods was refactored more than the code with God Classes, Data Classes, and Long Parameter List. The authors conjectured that Feature Envy was easier to refactor than God Class. Data Classes and Long Parameter Lists were not considered problematic by the developers and were not refactored. They found, similar to the authors in Ref. [22], that the code smell instances introduced in early revisions and removed in one of the next few revisions were typically removed as side effects of other maintenance tasks. The authors concluded that there is a general lack of awareness or concern in relation to code smells and that the developers seldom conduct tasks just for the purpose of removing code smells.

Counsell et al. [26] reported several studies that support the above findings. The studies are based on analyses of five open-source Java systems and a subsystem of a proprietary C# web-based application. The main finding is that the assumed effort to remove a code smell is a key factor for developers in their decision to remove this code smell or not. Göde and Harder [47] investigated how often changes occurred in a duplicated code. They found that consecutive changes to a duplicated code tend to occur and that the majority of these changes were *consistent* (i.e., simultaneous updates were made in the original and the duplicated code) or were intentionally inconsistent. They also found that "unwanted" inconsistencies led only to defects with low severity.

2.4 The Knowledge Gap in Code Smell Research

The effort by the code smell research community has, to a great extent, focused on the formalization and detection of code smells. In the current body of knowledge, few empirical studies shed light on the actual impact of code smells on software maintenance. Code smell detection tools can aid in the detection and measurement of code smells. However,

their interpretation and refactoring-related decisions still rely on expert judgment due to the lack of knowledge on quantifiable relations between code smells and maintainability.

From the identified empirical studies in code smells, it is possible to observe that not all code smells are equally harmful. Also, code smells are not harmful to the same extent over different contexts, indicating that their effects are potentially contingent on contextual variables or interaction effects. For example, Li and Shatnawi [77] found that the presence of Shotgun Surgery leads to defects. D'Ambros et al. [28], on the other hand, found no such connection between Shotgun Surgery and defects. Results from studies on Duplicated Code suggest that the effects of duplication depend on factors such as the programming language; e.g., the results from the COBOL system differed from that of the other types of systems in the study by Juergens et al. [63]. Similarly, results from studies on God Class seem to give different results. Deligiannis et al. [30] reported that the presence of a God Class indicates problems, while Abbes et al. [1] concluded that a God Class in isolation is not harmful. Olbrich et al. [108] reported that God Class is less connected with problems when adjusting for differences in file size, i.e., when file size was used as an independent variable in the regression model.

One reason for the difficulty of integrating and interpreting the results may be the variations in the dependent (outcome) variables, i.e., the variables used to represent maintainability. In the current studies on code smells, there are basically two categories of dependent variables: *effort* (the amount of time spent or the size of the changes required to finish the tasks) and *quality* (the presence or number of defects in the resulting product). Only Deligiannis [31] measured effort as the actual work effort spent on maintenance, which was recorded on video. Lozano et al. [80] and Olbrich et al. [108] used, instead of a direct measurement of effort, measures related to change frequency, change impact, and change size. However, it is questionable how good these surrogates are for measuring effort, and these studies do not refer to other studies that have validated or investigated the relationships between actual maintenance effort (time) and its surrogates. Quality is also measured using different measures in the previously reported empirical studies. Deligiannis [31, 30] used measures related to correctness, completeness, and consistency as quality indicators. The remaining studies used the number of defects per class or line as measures. Monden [100] used the number of revisions as a quality-related variable, arguing that a module is, on average, less maintainable the more times the module has been revised.

An additional reason why the empirical results may be hard to interpret is the variations in the context of the studies and in the research methods applied. The context of the studies varies with respect to the domain of the system and its size, the type of task and its size, the characteristics of the developers, and the code smell detection procedure.

The research methods and contexts of the studies reported in this review include one controlled experiment (Deligiannis [30]) in a context with students and relatively small tasks, one case study (Deligiannis [31]) in an academic context, two case studies that analyzed the existing code in commercial systems (Monden [100] and Juergens [63]), and several post hoc correlation and regression studies involving OSS projects [69, 77, 107, 108].

OSS projects have opened a new arena for post hoc correlation and regression studies in software engineering. However, the ability to claim cause-effects and to explain the results may be limited in such studies due to the inaccessibility of much of the process and context information that affects the outcomes of a software project.

Despite the plethora of detection methods and analysis tools, very little is reported on how code smells actually perform when conducting maintainability assessments. Studies seldom address the applicability of code smells in industrial, real-life contexts and validate essential aspects such as their descriptive richness or their capability to assess different maintainability factors.

We believe that more empirical studies are needed to support refactoring decisions and maintainability assessments based on code smells. Moreover, we think that more consistent operationalizations of maintainability constructs would ease the level of comparability across studies. Last but not the least, we believe that more in vivo studies are required, involving professionals, realistic maintenance tasks, and industry-relevant systems, to ease the transfer of results from a study to the software industry. Realistic study contexts, in spite of being more difficult to attain, are likely to enable higher confidence in the results and lead to more practical insights to both academia and industry.
The present research attempts to address segments of the identified knowledge gap and to improve the use of code smells in industrial software maintenance contexts. For this purpose, we take different perspectives of maintainability into account. Outcome-based interpretations of maintainability (e.g., effort, defects, and change size) are considered, but we also include qualitative process-related aspects (e.g., number and types of maintenance problems and developers' perception of maintainability) to better understand the underlying mechanisms of the code smell effects.

2.5 Software Maintainability and Its Operationalization

Maintainability is one of the software qualities defined by the ISO standard [59](p.10): "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications." To operationalize this high-level definition

of software quality, numerous surrogate measures and software measurement frameworks have been suggested. Recent work on quality models, with a focus on maintainability, can be found in Refs. [64, 118], as well as in the technical standards defined by ISO [60]. Oman [110], for example, suggested a taxonomy for maintainability measures, and Kitchenham et al. [70] defined an ontology for software maintainability, with the goal of identifying and describing the major maintenance factors influencing the software maintenance processes.

The ontology of Kitchenham et al. proposed four domain factors that influence the maintenance processes: product, organizational process, maintenance activities, and people. Within software engineering research, relatively high emphasis has been given to the *product* and *process* factors, e.g., in the context of estimating maintenance effort or maintainability assessments. Examples of process-centered approaches for maintenance effort estimation include those presented in Refs. [48, 75, 122, 76, 36]. Many of the process-centered approaches utilize process-related metrics or historical data to generate estimation models. Examples of product-centered approaches for estimating maintenance effort or assessing maintainability (as well as assessing other maintainability-related aspects, such as program comprehension and fault proneness) include those discussed in Refs. [109, 106, 34, 91, 104, 35, 150, 5, 136, 13, 55].

Other approaches within the identified literature suggest analyzing both process- and product-related aspects. This is the case with Mayrand [92], who combined the capability assessment (based on the ISO/IEC-12207 standard) and static analysis. Rosqvist et al. [127] combined static analysis with expert judgment. Other examples of "hybrid approaches" are found in Refs. [25, 41, 3, 29]. A literature review on maintenance cost estimation models is provided in Ref. [71], and one on maintainability assessment and metrics can be found in Ref. [124].

Despite the diversity of approaches for software maintainability assessment and maintenance effort estimations, there is a general consensus in relation to software product characteristics having a potentially strong impact on the ease of the maintenance work (Banker [11], p.434). Thus, it is fundamental to understand the structures and interactions of a software product and how they hinder or facilitate its change and improvement.

2.6 The Predominant Challenge in Software Maintainability Assessments

A major challenge in product-based maintainability measurement and evaluation is establishing validated relationships between the measurable attributes and the quality attributes of interest. This constitutes a challenge because the effects of the measurable

properties on the quality attributes may, as reported earlier, depend on the context of the system. Given that software maintainability is a context-dependent attribute, constructs representing maintainability should ideally be adapted to a given context of evaluation. For instance, measures operationalizing different constructs should have different weights or priorities according to the goals that the organization holds with respect to the system under analysis.

The incorporation of contextual information (e.g., pertaining to a certain maintenance process) is essential to give an accurate and useful interpretation of measures describing software design. That being so, there is a need to provide methodological support for building, adapting, and validating such models for a given context or setting.

One possible approach to build and adjust quality models (e.g., maintainability models) is by using expert judgment. Jørgensen [61] reported that combining expert assessment and formal methods usually provided the best results for software effort estimations. However, a structured set of steps should ideally be in place to enable such combinations for building maintainability models. Methodologies or techniques for combining expert judgment and formal analysis have been developed for domains other than software engineering. However, these techniques need to be adapted to make them useful in our research context. In this thesis, we adapt Concept Mapping, a technique from social research, to a software maintainability assessment context. The adapted technique is described in detail and is suggested as a structured set of steps that can be applied to use expert judgment to build/adjust quality models based on product metrics such as code smells.

Chapter 3

Research Methodology

This section consists of two subsections (Sections 3.1 and 3.2). In Section 3.1, we describe aspects of case study research relevant to our research problem and provide argumentation for the choices made in the design of our study. In Section 3.2, we describe the design of the study.

3.1 Case Study Research

This section starts by discussing the foundations of the case study, which is the main research methodology used in this dissertation. Subsequently, it describes the advantages and limitations of this methodology and how some of the disadvantages were addressed in the design of the study. The section finalizes with arguments on how the proposed approach on case studies can pose notable advantages for the research problems addressed here.

3.1.1 Definition and Types of Case Study

A case study is an empirical inquiry or research strategy that "investigates a contemporary instance or phenomenon within its real-life context, particularly when boundaries between instance or phenomenon and context are not clear" [152] (p.23). This research methodology is commonly used in social sciences, and its usage has recently increased within software engineering research [128]. Case studies often cope with situations where there are several variables of interest, multiple sources of evidence, rich contexts, and less control of variables than in experimental studies.

A case study research can be based on both quantitative and qualitative evidence and encompasses a wide set of systematic techniques (i.e., techniques for data collection, analysis, and reporting of the results) [152]. This methodology can provide a deeper insight

into key aspects that can be investigated further to develop or confirm theories that can explain an observed phenomenon [37].

A case study research includes single-case studies (examining a single instance or event) and multiple-case studies (investigating multiple instances or events). Yin [152] distinguished between two types of design for single- or multiple-case studies depending on whether they have single (holistic) or multiple units (embedded) of analysis. When a single- or multiple-case study is selected as the research design, case selection often depends on the nature of the case, proviso they are key or critical cases, outlier or extreme cases, revelatory cases, or typical or representative cases. Multiple-case studies may be preferred because single-case designs suffer from risks of misinterpretation or lack of necessary evidence to reach robust and generalizable conclusions.

Depending on the type of research question, case studies could be of exploratory, descriptive, or explanatory nature. Runeson and Höst [128] mentioned an additional category of case studies called *improvement* as case studies in software engineering take an improvement approach similar to what happens with *action research* (e.g., Andersons and Runeson [8]). Runeson said that this category was also described by Robson [126] as "emancipatory" in the social science context.

3.1.2 Advantages and Disadvantages of Case Studies

George [45] stated that case studies have particular advantages in answering certain kinds of questions: (1) the potential to achieve high construct validity, (2) strong procedures for fostering new hypotheses, (3) causal mechanisms in the context of individual cases, and (4) the capacity to address causal complexity.

Construct validity: Constructs are often difficult to operationalize (e.g., maintainability and comprehensibility), and the usefulness of a given operationalization is contingent on variations in the context of the study. For instance, an operationalization of a concept, such as "readability," may be useful in one context but misleading in another. Case studies may benefit from a detailed consideration of contextual factors for the purpose of higher construct validity.

Fostering of new hypotheses: Case studies also have advantages in relation to the identification of new variables and the development of new hypotheses through the study of deviant or outlier cases via techniques such as interviews, archival research, and observational protocols. These techniques allow the identification of alternative theories from the field when outcomes are not explainable by initial theories.

Exploration of causal mechanisms: Case studies may be able to use a rich context to look into the causal mechanisms in detail for the cases investigated. The inclusion of a large number of variables may enable the observation of unexpected aspects of a particular causal mechanism or the identification of conditions that activate a given causal mechanism.

Addressing causal complexity: Case studies can deal with complex causal relations, such as equifinality[1] [27], complex interaction effects, and path dependency [120].

Alongside the benefits, George [45] listed some challenges of case studies: (1) problem of case selection bias, (2) difficulty in determining relative causal weights for variables, and (3) tradeoffs between achieving high internal validity and generalizations that apply to broad populations.

Case selection bias: In multiple-case studies, the units of analysis must be selected to have a variation in the properties that the study intends to compare. However, in practice, many cases are selected based on availability [12].

Relative causal weighs: Case studies are often better at assessing whether and how a variable mattered to the outcome rather than how much it mattered. For example, the equivalent of beta coefficients in statistical studies may only be possible in case studies when there is enough control so that extremely similar cases differ only in one independent variable.

Tradeoffs between theoretical parsimony and explanatory richness: Case studies imply a tradeoff between offering rich explanations of particular cases and providing a theory in general terms that can be applicable across different types of cases.

3.1.3 Addressing Case Study Limitations through Controlled Multiple-Case Studies

A possible approach to address some of the shortcomings of typical case studies is to design a case study with a higher degree of control,[2] with characteristics similar to those

[1] *Equifinality* is the principle that in open systems a given end state can be reached by many different paths or trajectories, suggesting that similar results may be achieved with different initial conditions and in many different ways.

[2] The use of control variables in case studies within software engineering was first reported by Salo and Abrahamson [130].

of controlled experiments. In that way, one can assemble a real-life situation where contextual factors across cases are made as similar as possible, and the variable of interest is made dissimilar. This approach differs from typical multiple-case studies because cases in the context of controlled multiple-case studies are to some extent designed rather than selected. This implies that there will be a tradeoff between the degree of realism and the degree of control needed in a controlled multiple-case study.

This study presents the following conditions:

- Four systems were available with identical functionalities and dissimilar code structures, i.e., different code designs and code smells.
- There was a real-world need (and available budget) for a maintenance project.
- A pool of developers with similar programming skills was available.

These conditions made it possible to build a real-life situation (a maintenance project) and to compare cases where programmers with similar skill levels would be able to perform the same maintenance tasks on systems with different code designs. This study design enables a reasonable control of the maintenance tasks and the programmers' skills, thus addressing the "relative causal weights" limitations to a certain extent. The realism of the context would help to address the external validity of the results. The ability to compare systems differing mainly in one variable (code design), together with the fact that the study involves a comprehensive observation of a realistic context, addresses the problem of "tradeoffs between theoretical parsimony and explanatory richness" to a certain extent.

3.1.4 Potential Benefits of Controlled Multiple Case Studies in Code Smell Research

In Section 2.4, the current gap in code smell research is described. Some of the points mentioned were the following:

- Lack of empirical studies on the relation between the code smells and the different software maintenance aspects
- Difficulty in interpreting the results from the empirical studies, given the variations in the measures and the research methods applied
- Lack of in vivo studies and studies involving qualitative approaches
- Limited accessibility to process/context information that could explain results from the studies

The design of the present study

- Constitutes an empirical investigation of code smells and software maintenance
- Comprises a reasonable level of control over essential contextual factors (e.g., maintenance tasks, programmers, programming environment, and process), thus facilitating the comparison and interpretation of results across cases
- Comprises an in-depth observation of real-life software maintenance work, which enables us to derive causal models of the variables of interest
- Provides detailed information on process/context information (descriptive richness) that can help provide a better understanding of the results from the study

In addition, in Section 2.6, it is argued that one of the challenges in evaluating and investigating software maintainability is its context-dependent nature. Given that case study research is particularly designed to cope with situations "when boundaries between instance or phenomenon and context are not clear" [152] (p.23), our study design may provide a better understanding of maintainability as a dynamic phenomenon.

In the software engineering arena, the popularity of case studies is recent, and its practices may still be more immature than other disciplines (such as social research and Information Systems research) [128]. Runeson argued for the adequacy of case studies in the software engineering field, given that software engineering is a field involving disciplines where case studies are normally conducted (e.g., social and political sciences). Hannay et al. [50] asserted that theory-driven research is not yet a major issue in empirical software engineering and referred to several articles that commented explicitly on the lack of relevant theory. Case study research could significantly contribute to the development of theories from observations in relevant fields and contexts.

3.2 Controlled Multiple-Case Study Design for Investigating Code Smells

This section provides an overview of the essential elements of the study and the study design presented in this thesis. The overall study design is aimed at empirically investigating the usefulness of code smells to support assessments of software maintainability in the software industry. First, we present the context of the study. Second, we provide an overview of the study design. Third, we provide descriptions of the variables, data sources, data collection techniques, and measurement extraction procedures. Finally, we provide a description of the analysis technique conducted to answer each of the research questions.

3.2.1 Context of the Study

This section describes the systems investigated and the details concerning the maintenance project (i.e., tasks, developers, location, project activities, and development tools).

Systems under study: The Software Engineering Department at Simula Research Laboratory sent out a tender in 2003 for the development of a new web-based information system to keep track of their empirical studies. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all using the same requirements specification. More details on the initial project can be found in Ref. [7]. The four functionally equivalent versions are designated as Systems A, B, C, and D. The systems were primarily developed in Java and had similar three-layered architectures but had considerable differences in their design and implementation. This is reflected in Table 3.1, which displays the physical lines of code (LOC) for all the different types of files in the system (Java, Jsp, and other files, such as XML and HTML).

Table 3.1: LOC per file type for all four systems.

System	A	B	C	D
Java	8205	26679	4983	9960
Jsp	2527	2018	4591	1572
Others	371	1183	1241	1018
Total	11103	29880	10815	12550

Their cost also differed notably (see Table 3.2) as the companies that were hired also differed notably in their bids. The bid price may have been affected by business factors within the companies (e.g., different business strategies to profit from a project; some companies were willing to bid low to enter a new market) but may also reflect differences in skill and coding quality. As reported in Ref. [7], the choice of parallel developments by four companies was a result of design issues (e.g., having a sufficient number of observations or projects) and practical issues (e.g., having sufficient financial means to hire the companies and human resources to observe their projects).

Table 3.2: Development costs for each system [7].

	System A	System B	System C	System D
Costs	€25,370	€51,860	€18,020	€61,070

The main functionality of the systems consisted in keeping a record of the empirical studies and related information at Simula (e.g., the researcher responsible for the study,

participants, data collected, and publications resulting from the study). Another key element of the functionality was to generate a graphical report on the number of different types of studies conducted per year. The systems were all deployed over Simula Research Laboratories' Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to a database in the CMS to access data related to researchers at Simula as well as information on the publications therein. During the operational stage of all four systems, the defects and change requests were recorded.

Maintenance project: The study is based on a real maintenance project, which had specific maintenance goals. In 2008, Simula introduced a new CMS called *Plone* [119], and it was no longer possible for the systems to remain operational. This provided the motive to conduct and investigate a maintenance project. The functional similarity of the systems allowed the investigation of cases with very similar contexts (e.g., identical tasks and programming language and similar development environments), and the differences in the systems' code design allowed us to observe the effect of the variable of interest (i.e., the presence of code smells). To conduct this maintenance project, developers from two software companies were outsourced at a total cost of 50,000 Euros.

Table 3.3 shows the tasks implemented during the project. The first two tasks consisted of adapting the system to the new platform, and the third task consisted of adding a new functionality. It was assumed that the adaptive and perfective tasks would introduce new defects and hence would include work on corrective tasks (i.e., the study also indirectly covers corrective tasks).

Table 3.3: Maintenance tasks carried out during the study.

No.	Task	Description
1	Adapting the system to the new Simula CMS	The systems in the past had to retrieve information through a direct connection to a relational database within Simula's domain (information on employees at Simula and publications). Now Simula uses a CMS based on the Plone platform, which uses an OO database. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications; a String type is used now. Task 1 consisted of modifying the data retrieval procedure by consuming a set of web services provided by the new Simula CMS in order to access data associated with employees and publications.
2	Authentication through webservices	Under the previous CMS, authentication was done through a connection to a remote database using authentication mechanisms available at that time for the Simula web site. Task 2 consisted of replacing the existing authentication by calling a web service provided for this purpose.
3	Add new reporting functionality	The devised functionality provided options for configuring personalized reports, where the user could choose the type of information related to a study to be included in the report, define inclusion criteria according to researchers who were in charge of the study, sort the resulting studies according to the date that they were finalized, and group the results according to the type of study. The configuration should be stored in the system's database and only be editable by the owner of the report configuration.

Developers and location of the study: The above described maintenance tasks were conducted individually by six developers from two software companies, one located in the Czech Republic and another located in Poland (three developers from each company). The developers were recruited from a pool of 65 participants in a previous study on programming skills [15], which also included maintenance tasks. All the selected developers had been evaluated to have a good and, perhaps more important for our study, a similar level of development skill. The skill scores of the instrument were derived from the principles in Ref. [16], where the performance on each task was scored as a structured aggregate of the quality (or correctness) and time for a correct solution for each task. More details on the skill score can be found in Ref. [15]. Developers were also selected based on their availability, English proficiency, and motivation for participating in the study. The project was conducted between September 2008 and December 2008 at the companies' sites (four weeks in the Czech Republic and three weeks in Poland. The author of this dissertation was present in both sites during the entire duration of the project, acting as a Simula representative and conducting the study.

Project activities, development environments, and tools: Initially, the developers were given an overview of the project (e.g., the maintenance project goals and project activities). They also completed a questionnaire and a set of programming exercises to familiarize themselves with the domain of the systems. A specification was given to the developers for each maintenance task, and when needed, they discussed it with the researcher present at the site. An acceptance test was conducted once all the tasks were completed for one system. The test cases were based on different scenarios that considered the following aspects: functionality, performance, browser compatibility, and security. For each scenario, the following checklist was used to ensure the correct functioning of the product: internal and external links, absence of broken links, field validation, error message for wrong input, validation of mandatory fields, database integrity, database volume robustness (e.g., big attachment files), access to functionality according to the user role, and browser compatibility (i.e., Internet Explorer, Firefox, and Safari). The defects identified during acceptance testing were recorded in an issue tracking system and in the acceptance test reports. In addition to the acceptance test, an open interview was conducted (individually) where the developer was asked about his/her opinion on the system(s) on which he/she had worked so far. The development tool used was MyEclipse [44]. The database consisted of local MySQL [111] servers configured in each of the developers' machines. Apache Tomcat [138] was the web server used, and it was also configured in the developers' machines. Trac [33] was the defect tracking system (similar to Bugzilla) used to report defects encountered or introduced during the maintenance tasks. Subversion or SVN [137] was used as the versioning system.

3.2.2 Overview of the Study Design

The study design addresses two aspects relevant to current research in code smells: It enabled (1) an in-depth investigation of (2) an industrial software development context. As mentioned in Section 2.4, studies on the effects of code smells in the current literature predominantly consist of correlation studies conducted on post-development projects on OSS or controlled experiments. The correlation studies typically do not allow for an in-depth understanding of causal effects due to limited contextual information on the projects. Moreover, controlled experiments have often been conducted on small systems and tasks, which means that the results may be difficult to generalize to typical industrial software development contexts.

Having four functionally equivalent systems with different code smells enabled the design of a multiple-case study, with software development tasks embedded within almost identical maintenance contexts and differing in the variable of interest: code smells (code design). The design of the study therefore, to some extent, enables control over the moderator variables, such as system functionality, tasks, programming skills, and development technology, to better observe the relations between *code smells* and different maintenance aspects, such as effort, defects, change size, and maintenance problems (see Figure 3.1).

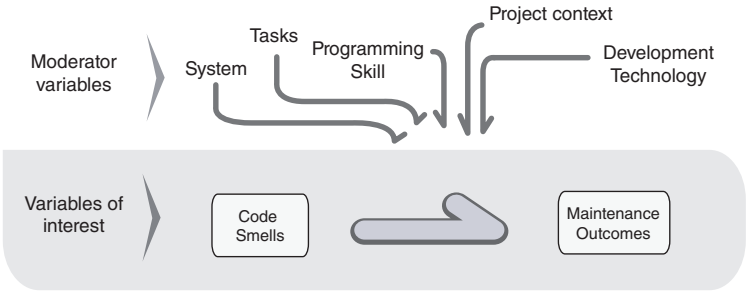

Figure 3.1: High-level design of the case study.

In multiple-case designs, it is possible to perform two types of replication: theoretical replication and literal replication. According to Yin [152], in *literal replication*, cases that are similar in relation to certain variable(s) are expected to support the analysis of each and give similar results. When *theoretical replication* is used, the cases that vary on the key variable(s) are expected to have different results.

Figures 3.2 and 3.3 illustrate the use of case replication for instances with similar and dissimilar presence of code smells. The developers in the study were required to perform the maintenance tasks in more than one system, so that there would be enough cases to

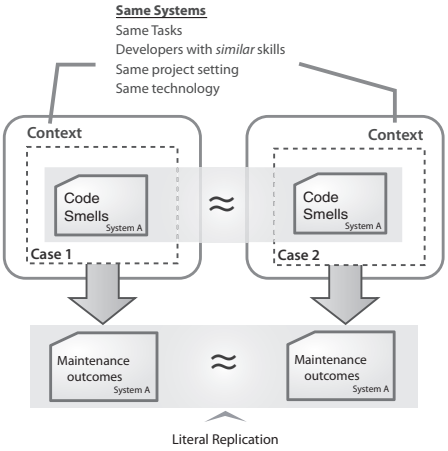

Figure 3.2: Illustration of how literal replication was applied in the study.

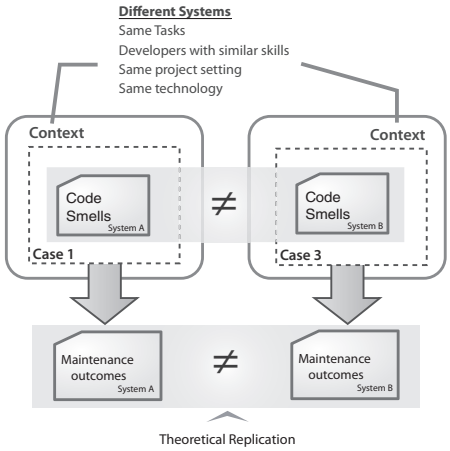

Figure 3.3: Illustration of how theoretical replication was applied in the study.

perform both types of replications. Note that if only theoretical replication was required, it would have been sufficient to have four different maintenance projects, each involving one system, and then to compare the results across the four systems.

As the use of both theoretical and literal replications can support each other's analyses, each of the six developers was asked to first conduct all tasks in one system (in the order that they were presented in Table 3.3) and then to repeat the same maintenance tasks on a second system, resulting in 12 observations (six developers × two systems). Thus, we make a distinction between first-round cases and second-round cases. "First round" denotes a case in which a developer has not maintained any of the systems previously, and "second round" denotes a case in which developers repeat the tasks on a second system. Figure 3.4 describes the order in which the systems were assigned to each developer. This

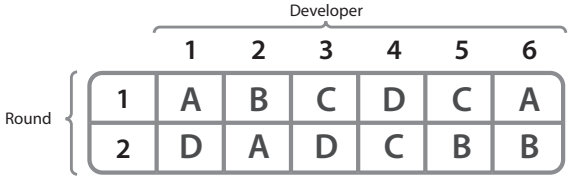

Figure 3.4: Assignment of systems to developers in the case study.

assignment was done randomly. The order (and thus combination) of systems maintained by each developer was decided so that all the four systems should be equally represented within the cases and so that all the four systems should be maintained at least once in

the first round and once in the second round (i.e., to support the adjustment for potential learning effects from the first to the second round).

3.2.3 Variables, Data Sources, Data Collection Activities, and Measurement Procedures

Figure 3.5 describes the moderator variables (those we control in the analysis), the variables of interest (those whose relationships we analyze), and the data sources for the variables. The variables of interest within this study are as follows:

1. *Code smells*: *Number of code smells* and *code smell density* (code smells/kLOC) are used as measures for this variable.

2. *Developers' perception of the maintainability of the systems*: This includes subjective and qualitative aspects of maintainability to be reported by the developer once the three maintenance tasks of one system had been completed.

3. *Maintenance problems encountered by the developers during maintenance*: These include a qualitative aspect of the maintenance process based on problems reported through interviews or think-aloud sessions or observed by the researcher during the maintenance work.

4. *Change size (measured by churn)*: This variable constitutes an outcome variable from the maintenance project, reflecting the sum of LOC added, changed, and deleted.

5. *Effort (measured in time spent on the tasks)*: This variable is an aspect of the maintenance outcome.

6. *Defects introduced during maintenance*: This variable is an aspect of the quality of the system after the tasks were completed. As such, it is also an aspect of the maintenance outcome.

Figure 3.5 discriminates between outcomes/aspects that were observed at the system level (one asterisk) and at both system and file levels (two asterisks). The figure also distinguishes maintenance problems and maintainability perception–which are categorized as *qualitative aspects* (circles)–from change size, effort, and defects–which are categorized as *quantitative outcomes* (squares).

Figure 3.5 also depicts the data sources from which each of the maintenance outcomes/aspects was derived. We conducted a comprehensive collection of both qualitative and quantitative data. Table 3.4 displays the complete list of data sources and their

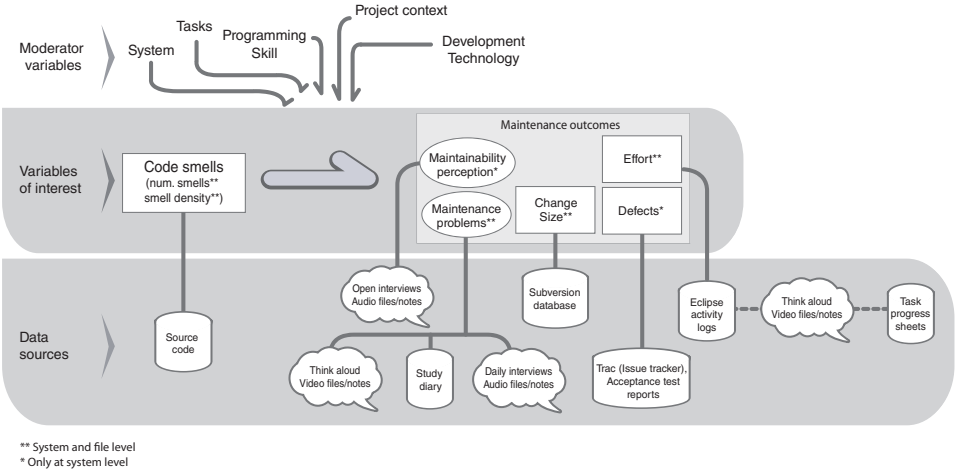

Figure 3.5: Illustration of variables involved in the study and the corresponding data sources.

corresponding data collection activities. In the remainder of this section, a description is provided on how the variables of interest were measured and extracted from the data sources.

1) Collection of information about code smells: Two commercial tools were used (Borland Together® [17] and InCode [58]) to detect code smells. This was done to facilitate the repeatability of the analysis as several studies have used Borland Together to investigate the effects of code smells (e.g., Li and Shatnawi [77]). Many studies report code smells detected by their own tools/methods, which complicate replication studies unless the detection strategy is well described. Another reason for our use of the two commercial tools was that they use the detection strategies (metrics-based interpretations of code smells) proposed by Marinescu [87], which seem to be relatively well known as acceptable strategies within the code smell research community.

Table 3.5 presents the list of code smells that were detected in the systems, alongside their descriptions, which are taken from Ref. [40]. Borland Together detected more code smells than those included in the table, but they were not included because it was not possible to find any instances of such code smells in the four systems we studied. The last code smell in the table, which we did collect information about, is traditionally not considered to be a code smell but rather a design principle violation, as described by Martin [90]. It was, however, treated as a code smell in our research because it constitutes

Table 3.4: List of data sources in the study.

No.	Data sources	Data collection activity
1	Source code	The system's source code. (Note: The scope of the code smells is limited only to Java files, and consequently, jsp, sql files and other artefacts were not considered for measurement).
2	Open-ended interviews Audio files/notes	Individual *open-ended interviews* (40-60 minutes): were held after all three maintenance tasks were completed for each system. The developers were asked about their opinion of the system(s) on which they had worked so far (e.g., *how difficult was it to understand the systems?*).
3	Daily interviews Audio files/notes	Individual *progress meetings* (20-30 minutes): were conducted daily between all developers and the researcher present at the study to keep track of the progress, and register difficulties encountered during the project (ex. Dev 1: "*It took me 3 hours to understand this method...*")
4	Subversion database	The repository database where the source code was kept (Subversion was used).
5	*Trac* (Issue tracker)	The developers registered defect reports in an issue tracking system called *Trac*.
6	Acceptance test reports	Results from each of the acceptance test scenarios
7	Eclipse activity logs	Developer's activities were logged by a plug-in called Mimec [74]. This plug-in logged all the actions performed on Eclipse at the GUI level.
8	Think-aloud video files/notes	Video-recorded *think-aloud sessions* (ca. 30 minutes) were conducted every second day to observe the developers in their daily activities. If the developer felt uncomfortable "speaking-out", the session would be limited to record the screen and the responsible of the think-aloud session will take observational notes.
9	Task progress sheets	The developers filled in these sheets, where they compared estimations vs. actual time for each of the sub-tasks required for the maintenance.
10	Study diary	A logbook was kept by the researcher present at the study where the most important aspects of the study were annotated in a daily basis

an anti-pattern believed to have negative effects on maintainability [90].

2) Collection of perceived maintainability: Once a developer finished all the three tasks for one system, he/she was asked to give his view on the overall maintainability of the systems through an open-ended interview (see source 2 in Table 3.4). The audio files from this interview were summarized by using a tool called Transana [149].

3) Collection of maintenance problems: A list of maintenance-related problems was extracted from the daily interviews, think-aloud sessions, and study diary (sources 3, 9, and 10 respectively, in Table 3.4). In the context of this study, a maintenance-related problem is interpreted as: *any struggle, hindrance or problem developers encounter, and observed through daily interviews and think-aloud sessions, while they performed their maintenance tasks.* The daily interviews with each developer enabled the recording of problems encountered while the problems were still fresh in their minds. The following is an example of a comment given by one developer who complained about the complexity

Table 3.5: Analyzed code smells and their descriptions, from Refs. [40] and [90]

Smells	**Description**
Data Class	Classes with fields and getters and setters not implementing any specific function
Data Clumps	Clumps of data items that are always found together whether within classes or between classes
Duplicated code in conditional branches	Same or similar code structure repeated within a the branches of a conditional statement
Feature Envy	A method that seems more interested in another class other than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.
God (Large) Class	A class has the God Class smell if the class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles.
God (Long) Method	A class has the God Method code smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method
Misplaced Class	In "God Packages" it often happens that a class needs the classes from other packages more than those from its own package.
Refused Bequest	Subclasses do not want or need everything they inherit
Shotgun Surgery	A change in a class results in the need to make a lot of little changes in several classes
Temporary variable is used for several purposes	Consists of temporary variables that are used in different contexts, implying that they are not consistently used. They can lead to confusion and introduction of faults.
Use interface instead of implementation	Castings to implementation classes should be avoided and an interface should be defined and implemented instead.
Interface Segregation Principle Violation	The dependency of one class to another one should depend on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling

of a piece of code: "It took me 3 hours to understand this method..." Comments like this were used as evidence that there were maintenance (understandability) problems in the file that included this method. The interviews were transcribed and summarized using Transana [149].

Think-aloud sessions were also used as a means to identify problems. During the think-aloud sessions, the developers' screens were recorded with ZD Soft Screen Recorder [154]. Sometimes, the maintenance problems were derived from more than one data source (e.g., by a combination of direct observation, the developers' statements on a given topic/element, and the time/effort spent in an activity). An example of the process followed to collect and structure data related to maintenance problems is given in Table 3.6. In this example, the observations by the researcher and the statements from the developer lead to the conclusion that the initial strategy of replacing several interfaces to complete task

1 was not feasible due to unmanageable error propagation. The developer spent up to 20 minutes trying to follow the initial strategy (i.e., replace the interfaces) but decided then to roll back and to follow an alternative strategy (i.e., forced casting in several locations) instead. As a result of this information (i.e., problems due to change propagation), the files containing these interfaces were deemed problematic.

Table 3.6: Excerpt from a think-aloud session.

Code	**Statement/Action by Developer**	**Observation / Interpretation**
Goal	Change entities' ID type from Integer to String	This is part of the requirements for Task 1.
Finding	"Persistence is not used consistently across the system, only few of them are actually implementing this interface so..."	Persistence[3] is referred to as two interfaces for defining business entities, which are associated with a third-party persistence library, and is not used consistently in the system.
Strategy	"I will remove this dependency, I will remove two methods from the interface (getId an setId) added for integer and string. This strategy forces me to check the type of the class, but this is better than having multiple type forced castings throughout the code."	Developer decides to replace two methods of the Persistence interface (i.e., getId() an setId()), which are using Integer, with methods with String parameters.
Action	Engages in the process of changing id in interface PersonStatement.java	Developer engages in the initial strategy.
Muttering	"Uh, updates? just look at all these compilation errors..."	Developer encounters compilation errors after replacing the methods in the interfaces.
Action	Fix, refactor, correct errors.	Starts correcting the errors.
Strategy	"Ok... I need to implement two types of interfaces, one for each type of ID for the domain entities. I will make PersistentObjectInt.java for entities that use Integer IDs and PersistentObject-String.java for String IDs."	Change of strategy, decides to actually replace the interface instead of replacing the methods in the interface.
Action	Fix more errors from Persistable.java.	More compilation errors appear.
Action	Continue changing interface of the entity classes into PersistentObjectInt and PersistentObjectString.	Attempt to continue with the second strategy.
Action	(After 20 minutes) Rollback the change.	Developer realizes that the amount of error propagation is ummanegeable so rolls back the changes.
Muttering	"Hmm... how to do this?"	Developer thinks of alternative options.
Strategy	"Ok, I will just have to do forced casting for the cases when the entity has String ID."	Developer decides to use the least desirable alternative: forced type castings whenever they are required.

[3]A persistence framework is used as part of Java technology for managing relational data (more specifically data entities). For more information on Java persistence, see www.oracle.com

During the interviews and think-aloud sessions, a logbook or study diary was kept to record maintenance problems in detail. For each identified maintenance problem, the following information was extracted:

a. The developer and the system

b. The statements provided by the developers related to the maintenance problem

c. The source of the problem, e.g., whether it was related to the Java files, the infrastructure, the database, the external services, etc.

d. List of files/classes/methods mentioned by the developer when talking about the maintenance problem

In short, the categorization of the problems and the identification of problematic files were based on either the direct observation of the developers' behavior in the think-aloud sessions or on the comments made by the developers during the daily interviews.

4) Collection of change sizes: As part of our attempt to understand the ways in which code smells affected maintainability, the Java files that were modified during maintenance were analyzed by observing their *churn*. Hall and Munson [49] defined a code churn as the absolute number of changes (i.e., number of lines changed + added + deleted) made over a number of versions of a software unit. This measure has been used in previous studies for analyzing the impact of code smells and other characteristics of a code [105, 107, 153]. This variable was calculated by writing a Java program that used SVNKit [139], a Java library for extracting information from Subversion.

5) Collection of maintenance effort: Eclipse activity logs generated by Mimec [74] were used as a data source for measuring effort because they allowed us to measure the exact time the developers spent on every Java file. The task progress sheets (source 9 in Table 3.4) completed by the developers provided a measure of the overall maintenance effort of a developer on a system.

The use of activity logs generated by Mimec was chosen to measure effort because Mimec provided high accuracy for the capture of events occurring in Java files. Mimec can capture Eclipse IDE events, such as editing source files (Java files), scrolling the source code window, switching between open files, expanding/collapsing trees in the package explorer, selecting Java elements (classes, methods, and variables), and running Eclipse "commands" (e.g., copy, save, and go to end of line). In the present study, the activity logs were stored as Comma-Separated Value (CSV) files, where every single line corresponds

Table 3.7: Description of data contained in an event.

1.	*Timestamp*	*Time* (in milliseconds) when the event was recorded
2.	*Date*	*Time* the event was observed by Mimec (very similar to #1)
3.	*Kind*	*Kind* of event: *edit*, *selection*, *command* or *preference*
4.	*Target*	A Java element (if any) that is the subject of the interaction, such as the name of the file selected, or the name of the class/method being edited.
5.	*Origin*	The part of Eclipse that generates the interaction (e.g., Package Explorer, Editor)
6.	*Delta*	An attribute (if any) containing relevant meta-information.

to an event, a single observation generated by Mimec. Each event consists of six pieces of data, as depicted in Table 3.7.

A Java program was written to identify the elapsed time between the different activities by truncating the consecutive events annotated with the same kind of event, target, and origin. Because most events that were associated with the source code were registered via Mimec, this was assessed to be an accurate measure of the effort spent reading or updating Java files in most cases. However, a heuristic was needed to handle two particular situations. In the maintenance project, the developers had to work with multiple environments besides the Eclipse IDE. They had to

- Look at documentations
- Run the application and interact with the GUI (website) component of the systems
- Work on the DB via tools other than Eclipse

For all these events, the developer would leave the IDE. Mimec does not record when the developer leaves the IDE but does so only when they return to the IDE. The algorithm calculates the time spent on the different activities by aggregating the elapsed time of consecutive log entries that marked with the same activity. When the activity in a log entry changes, the aggregation starts over.

As a result of the previous situation, the elapsed time from the moment that any developer leaves the IDE until he/she comes back will be assigned to the activity performed just before leaving the IDE; this will yield inaccurate results. For example, a developer may first select a file and then leave the IDE to take a coffee break, and all the time spent on the coffee break will be assigned to the action "select file." Another problem of Mimec is that for certain editing commands (e.g., copy, paste, and cut), it does not register the file in which the activity is done. The algorithm for calculating the time spent on a file works under the assumption that any log entry that involves a file will contain the filename related to the activity.

To solve the "idle time" problem, a lookup table was created with average times of all types of activities from all the logs of all developers. The average values excluded any log entries that occurred just before the activity "Go back to IDE" (because those are precisely the ones that we want to correct). Sample sizes used to compute those averages were very high, and standard deviations were very low, so they were considered trustworthy. The algorithm for calculating the time for consecutive activities was adjusted as follows. If any activity was followed by Go back to IDE, then the following would happen:

Case 1: If the elapsed time between entries is equal or lower than the average time indicated in the lookup table, assign the whole elapsed time to the entry.

Case 2: If the elapsed time between entries is higher than the average time indicated in the lookup table, assign the average time from the lookup table to the entry and the elapsed time minus average time to "Unknown activities outside IDE."

To solve the "missing file" problem, the filename contained in the closest preceding entry to any log entry containing any of the problematic commands (copy, paste, and cut) was used as the filename. This is because to perform any editing command in a file, it is necessary to select the file, and this event always captures the filename. With this solution, it is possible to ensure that all editing commands have a filename, and therefore the effort spent on files will be accurate. The adjustment for the missing file problem was performed after the analysis on Paper 2 was concluded. Without the adjustments, 298 Java files were identified as inspected/modified during maintenance. After the adjustments, 301 Java files were identified as inspected/modified during maintenance. This last set was used as part of the analysis in Paper 3. Given the low delta between the two data sets, it was decided that this would not affect the conclusions drawn in Paper 2, so no re-analysis was conducted.

6) Collection of defects: To measure the final number of defects introduced by the developer at the end of the tasks, the issue tracking system (Trac) and the acceptance test reports were consulted. The list of defects from production available in Ref. [7] was also considered in the total set of defects for each of the systems. To distinguish between defects originally existing in the system before maintenance and defects introduced during maintenance, we attempted to reproduce the defects in the original versions of the systems. The defects were categorized by the researcher present at the study and were weighted according to the *orthogonal defect classification* [23]. Ideally, the defects would have been associated with Java files, but because not all developers associated a defect with a change in a file, it was not possible to collect all the defects at the file level but only at the system level.

3.2.4 Data Analysis

Analyses addressing RQ1: Code smells were aggregated at the system level, and each system was ranked according to the *amount of code smells* they contained and their *code smell density* (i.e., less code smells and lower smell densities mean better maintainability ranking of a system). After the systems had undergone maintenance work, the maintenance outcomes: total effort, and the introduced defects at the system level were collected per system to rank them accordingly. To avoid the learning effect problems, we used only the data from the first round per developer. Cohen's kappa coefficient[4] was used to statistically measure the degree of agreement between the code-smell-based and maintenance-outcome-based rankings. Previous maintainability assessments of the systems based on a subset of C&K metrics and expert judgments, as reported in Ref. [6], were also compared with the maintenance-outcome-based rankings to analyze the differences in accuracy between the code-smell-based, expert-judgment-based, and metrics-based approaches for maintainability assessments.

Analyses addressing RQ2: This question was addressed by focusing on Java files as the unit of analysis and by using multiple regression analysis. Effort at file level (effort used to view or update a file) was the variable to be explained. Variables representing the different code smells, the file size (measured in LOC), the number of revisions on a file, the system, the developer, and the round were included as independent variables. Several regression models, with different subsets of variables, were built to compare their fit and to discern the predictive capability of each of the variables considered.

Analyses addressing RQ3: This question was addressed by focusing on Java files as the unit of analysis and by using binary logistic regression analysis. The variable to explain was the variable "problematic," which was *true* (1) if a file was deemed problematic during maintenance by at least one developer who worked with the file, but *false* otherwise (0). The different types of code smells, files size (measured in LOC), and change size (churn) were used as independent variables. A principal component analysis (PCA) using orthogonal rotation (varimax) was conducted on a set of files to observe patterns of collocated code smells. A follow-up qualitative analysis based on the data from the interviews and the think-aloud sessions was performed (1) to support/challenge the findings from the binary logistic regression and (2) to understand better how the presence of a code smell contributed to the problems experienced by the developers during maintenance.

[4]Cohen's kappa coefficient is a statistical measure to represent inter-rater agreement for categorical items.

Analyses addressing RQ4: This question was addressed through a compilation and synthesis of the relevant qualitative data related to problems encountered by the developers during the maintenance work. The record of the problems was based on observational notes, think-aloud sessions, and progress interviews. Based on the origin of a problem, each problem was categorized as a source-code-related or non-source-code-related. The extent to which code smells can explain the problems during maintenance was investigated by observing the proportion of problems that were related to the source code compared with the problems caused by other factors (e.g., problems related to infrastructure or external services). The set of problems associated with the source code was further investigated by examining how many of these problems could potentially be related to the presence of code smells. This was done by examining the presence of code smells in files related to maintenance problems.

Analyses addressing RQ5: This was addressed through a mainly qualitative analysis, which compared the developers' perceptions on the maintainability of the systems with the goal of identifying a set of factors relevant to maintainability. These factors were related to current definitions of code smells to observe their conceptual relatedness. The transcripts of the open-ended interviews were analyzed through *open* and *axial coding* [135]. The identified factors were summarized and compared across cases using a technique called *cross-case synthesis* [152]. The factors derived from this analysis were compared with the factors reported in a previous study [6], which were extracted via expert judgment.

Analyses addressing RQ6: This question was addressed through a descriptive case study with a detailed account of how concept mapping (a technique from social research) can be adapted to software engineering. This technique was suggested as a structured approach to enable the usage of expert judgment in guiding the selection, combination, and interpretation of code smells for maintainability assessments. Several software engineering researchers and a senior software engineer (with more than 25 years of experience at that time) participated as the experts in the concept mapping process. We compared the resulting concept maps (representing the maintainability of the four systems) with the results from the expert assessment reported in Ref. [6] to evaluate the validity of the concept mapping technique.

Chapter 4

Summary of Results

In this section, the key results of the papers submitted as part of this thesis are summarized.

4.1 Code Smells as System-Level Indicators of Maintainability

System-level indicators of maintainability based on code smells were investigated in four systems where a system's maintainability was ranked according to code smell measures and compared with respect to the maintenance outcome measures – change effort, and number of defects. Figure 4.1 shows the standardized values of the total number of code smells and the code smell densities for each system. When differences in code smells were not adjusted for differences in size (i.e., by using the total number of code smells rather than the code smell density), the smallest system (system C) was considered the most maintainable. Conversely, when code smell density was considered (which adjusts for the size of the systems), the largest system (system B) becomes more maintainable.

Figure 4.2 presents the standardized scores for both effort and defects. As can be seen, these two variables give similar ranks regardless of the use of effort or defects as our maintenance outcome measure. Figure 4.2 suggests that system C is the most maintainable, followed closely by system D. System A has an intermediate maintainability level, while system B is assessed to be the least maintainable system.

In Ref. [6], the code-metrics-based assessment approach resulted in system D being the most maintainable, systems A and B being medium maintainable, and system C being the least maintainable. On the other hand, the expert-judgment-based approach resulted in systems A and D as the most maintainable, system C as medium maintainable, and system B as the least maintainable. When comparing the rankings from all the

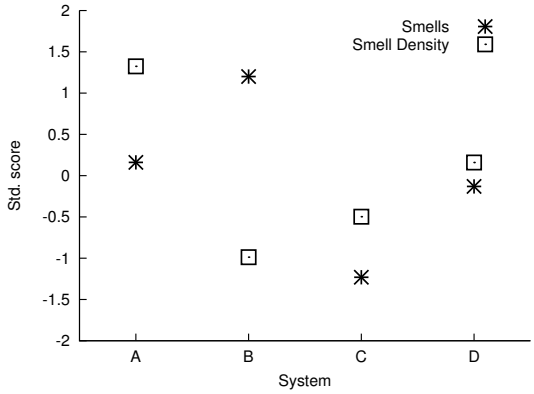

Figure 4.1: Standardized number of code smells and code smell densities.

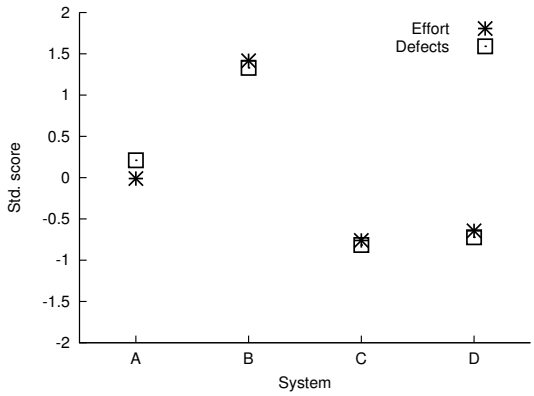

Figure 4.2: Maintenance scores for the systems

assessment approaches with the actual maintenance outcomes (see Table 4.1), the *number of code smells* gave the best matching [with a kappa coefficient (Po) of 0.75]. Conversely, the *code smell density* displayed zero matching with the maintenance outcomes. The evaluation approaches reported in Ref. [6] showed a medium level of agreement [kappa coefficient (Po) of 0.50].

Table 4.1: Comparison of levels of agreement with actual maintenance outcome.

Assessment approaches	**Level of matching or agreement**	**Kappa coefficient**
Code metrics from [6]	Matching D as most maintainable and A as intermediate.	0.50
Expert Judgment from [6]	Matching D as most maintainable and B as least maintainable.	0.50
Number of code smells	Matching C as most maintainable, A as intermediate and B as least maintainable.	0.75
Code smell density	No matching with maintenance outcomes	0.00

Code smell density analysis implied that system B (the largest system of all four) was highly maintainable, which is not an accurate assessment, at least for the size and types of maintenance tasks involved in the project. This result suggests that one should be careful when using code smell density to compare systems differing greatly in size. However, when only considering systems similar in size (i.e., when excluding system B), code smell density reflected better the levels of maintainability according to the outcomes in terms of effort and defects. To illustrate this, Figure 4.3(a) and 4.3(b) show *parallel plots*[1] of the standardized scores for the number of code smells and code smell density, effort, and defects for the three systems with similar sizes. This figure shows that the degree of correspondence between code smell measure and effort/defects is better for code smell density than for number of code smells.

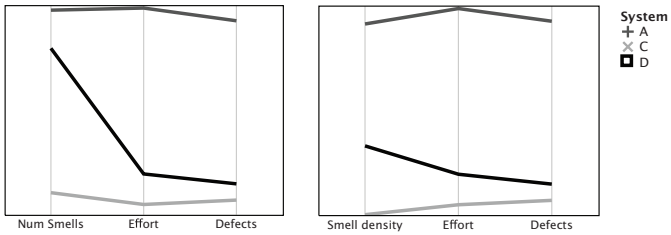

Figure 4.3: Parallel plots for systems A, C, and D on the level of matching between the standardized scores of (a) number of code smells and (b) code smell density versus maintenance outcomes, that is, effort and defects.

[1]Parallel plots show connected-line plots of several variables at once.

The figure suggests that the effect of code smell density tended to be sensitive to larger differences in system size but that the use of this measure would improve if systems of similar sizes were compared and might provide more information than just the sum of the number of code smells per system. When comparing code-smell-based assessments with other assessment approaches, the C&K metrics provided more insight into which system had the most "balanced design" (e.g., they pointed out the absence of deviant classes in a system), but this measure tended to ignore the effect of the task size when maintenance tasks were of small/medium size. Expert-judgment-based assessment was the most flexible of all the three approaches because it considered both the effect of the system size and the potential maintenance scenarios (e.g., small versus large extensions). We conclude that an advantage of the use of code smells is that when comparing similarly sized systems (i.e., with the use of code smell density), they can spot critical areas that experts may overlook.

4.2 Quantifying the Effect of Code Smells on Maintenance Effort

To quantify the effects of code smells on maintenance effort, a multiple regression analysis was conducted, with effort at the file level as the variable to explain. Twelve different types of code smells were used as variables to determine to what degree the presence of code smells explained the effort at the file level. The regression analysis tried to adjust for the effect of the variables: file size, number of revisions, system, developer, and round. Four different regression models were developed.

An R^2 of only 0.15 for model 0, which includes only system, developer, and round as variables, indicates that these variables did not explain much of the maintenance effort per file. Model 1 adds the variables related to the 12 code smells to model 0 and gives an R^2 of 0.36, with Feature Envy significant at $\alpha < 0.001$ and God Class significant at $\alpha < 0.01$. Refused Bequest was significant at $\alpha < 0.05$ but displayed a negative coefficient. Model 2 adds the variable file size to model 1, improving the R^2 to 0.42. The negative effect of the code smells on maintenance effort was reduced as Feature Envy was only significant at $\alpha < 0.05$, and the rest of the code smells display no longer significant increasing effects on effort. The positive effect (decrease in effort) for classes with Refused Bequest in this model was still significant ($\alpha < 0.01$). Model 3 adds the variable number of changes to model 2, which improves the R^2 to 0.58. The only code smell that remains a significant independent variable of effort ($\alpha < 0.01$) in the regression model is Refused Bequest, which still had a decreasing effect on effort. If we exclude the code smells from model 3, the R^2 stays at 0.58. This finding implies that code smells may not provide additional

explanatory power than file size and the number of revisions in the context of explaining effort usage per file. The regression analysis confirms that the file size and the number of changes are both significant ($\alpha < 0.001$) predictors of effort.

4.3 Investigating the Capability of Code Smells to Uncover Problematic Code

To investigate the capability of code smells to uncover problematic codes, binary logistic regression was conducted. We built a model in which the variables related to the 12 code smells, the file size, and the file churn were entered in a single step. The R^2 values (Hosmer & Lemeshow = 0.864, Cox & Snell = 0.233, and Nagelkerke = 0.367) confirm that our model provided a reasonably good fit of the data. The performance measures for the model are *accuracy* = 0.847, *precision* = 0.742, and *recall* = 0.377.

Files belonging to system B (the largest system of all four) displayed an odds ratio of 4.137. This suggests that the large size of the system might be quite important when explaining the number of maintenance problems. The odds ratio for the code smell ISP Violation was the largest [Exp(B) = 7.610, $p = 0.032$], which suggests that ISP Violation was able to explain much of the maintenance problems at the file level. The model also finds Data Clump as a significant contributor [Exp(B) = 0.053, $p = 0.029$], but contrary to ISP Violation, this code smell indicates less maintenance problems.

A principal component analysis (PCA) was conducted on the data points using orthogonal rotation (varimax). The code smells: God Method, God Class, and Temporal variable used for several purposes, Duplicated code in conditional branches, and Feature Envy belonged to the same factor (factor 1). The fact that detection strategies for God Class and God Method are based on size measures hints that this factor represents code smells that tend to appear together in large classes. ISP Violation and Shotgun Surgery belonged to the same factor (factor 2), which indicates that they may represent the same construct to some extent (e.g., related to widespread afferent coupling) and that they do not seem to be connected with large classes (as in factor 1). Data Class and Data Clump belonged to the same factor (factor 3). The code smells Implementation instead of interface seems to appear very seldom in our data set and did not cluster with any of the other code smells (factor 4). The code smells Refused Bequest and Misplaced Class belonged to the last factor (factor 5).

When examining the problematic files that contained at least one code smell, we observed that a large proportion of them (7 out of 12) contained both ISP Violation and the Shotgun Surgery but no other code smells. This further suggests that these two smells, either individually or together, may be essential to explaining a major part of the

maintenance problems. The analysis on the qualitative data exposed a relation between ISP Violation and some of the negative consequences entailed by widespread afferent coupling, more specifically the following:

1) Error propagation: Introduction of defects in classes with ISP Violation led to an error spread across different components of the system. This situation made much of the systems' functionality stop working after changes on the ISP Violators.

2) Change propagation: When changes were introduced to files containing ISP Violation, we observed that adaptations or amendments were needed on other classes depending on the ISP Violators. This situation often resulted in time-consuming change propagation.

3) Difficulties identifying the task context: The presence of cross-cutting concerns, which were captured by ISP Violation, made the identification of relevant task contexts difficult.

4) Confusion due to inconsistent design: Inconsistent and arbitrary allocation of data and functionality was considered a confounding factor by the developers who would not understand the rationale of the design. The situation resembled what Martin [90] described as a "wider spectrum of *dissimilar* clients." ISP Violation was able to capture this problem.

Finally, we found evidence of the existence of interaction effects between ISP Violation and other code smells and between ISP Violation and other types of design shortcomings in the software. These interaction effects were deemed to sometimes have serious consequences on the maintenance process.

4.4 The Coverage of Code Smells to Explain Maintenance Problems

To investigate how much of the overall maintenance problems were related to code smells, all the maintenance problems the developers encountered during the maintenance were identified and categorized according to their source. Further analysis was conducted to determine if the problem was related to the source code to assess the extent to which those problems could be explained by the presence of code smells. The majority of the problems turned out to be related to one of three situations: (a) introduction of defects as a result of changes (25%), (b) troublesome program comprehension and information searching (27%), and (c) time-consuming changes (39%).

In total, 137 different problems were identified. Out of these, 64 problems (47%) were associated with the Java source code of one the four systems. The remaining 73 (53%) are problems not directly related to the source code. The types of noncode-related problems were architecture, initial defects in the system, developer factor, external services, inadequate infrastructure, runtime environment, and requirement specification.

Problems associated with source code were divided into three groups: 37 problems were attributed to the presence of code smells (58%), 19 problems were attributed to other code characteristics (30%), and 8 problems were the result of the interaction between several code characteristics, some of them including code smells (12%).

Problems associated with characteristics other than code smells were semantic inconsistencies, internally complex files (in terms of the number of properties and methods), cyclic dependencies, external libraries, and implementation shortcomings. We identified four code characteristics that explained the maintenance problems and were reflected in existing code smells: (a) inconsistent design (including use of variables), (b) cross-cutting concerns, (c) large and complex classes, and (d) coupling (both afferent and efferent).

We found that 12% of the problems occurred because of the interaction effects between code smells or between a code smell and other code characteristics. Finally, we found that when code smells are distributed across files that are *coupled*, the consequences stemming from the interaction effects between these code smells are equivalent, from a practical perspective, to those between code smells *collocated* in the same file. Consequently, in practice, there is no difference between the interaction effects of coupled smells and the interaction effects of collocated code smells.

4.5 The Relationship between Code Smells and Maintainability Factors

This relationship was investigated through an analysis of data from the open-ended interviews. The maintainability factors identified in the analysis were compared with a previously reported set of maintainability factors, which were extracted from an expert-based maintainability assessment involving the same systems [6].

Thirteen factors were identified from the study: appropriate technical platform, coherent naming, design suited to the problem domain, encapsulation, inheritance, (proprietary) libraries, simplicity, architecture, design consistency, duplicated code, initial defects, and logic spread. The first nine factors coincided with the factors reported by experts in Ref. [6]. The last four factors are new factors identified through the study.

Appropriate technical platform, simplicity, and design consistency were the factors most mentioned by all developers. Standard naming conventions and comments, men-

tioned by experts in Ref. [6], were not important to the developers. The design consistency was considered as one of the most important factors because having a consistent schema for variables, functionality, and classes represents a clear advantage for code understanding, information searching, impact analysis, and debugging.

The analysis of the conceptual relatedness between the identified maintainability factors and the current definitions of code smells showed the difficulty of using code smells to analyze the total maintainability of a system. For example, the maintainability factors–appropriate technical platform, coherent naming, design suited to the problem domain, initial defects, and architecture–needed techniques other than the current set of code smells.

Nevertheless, there are code smells capable of supporting the analysis of the maintainability factors–encapsulation, design consistency, logic spread, simplicity, and use of components. For example, simplicity is a factor traditionally addressed by static analysis means, but it is also closely related to God Class, God Method, Lazy Class, Message Chains, and Long Parameter List. Similarly, logic spread is related to Feature Envy, Shotgun Surgery, and ISP Violation, and the design consistency factor (one of the most critical factors) is related to several code smells, such as Alternative Classes with Different Interfaces, ISP Violation, Divergent Change, and Temporary Field.

Of the 13 maintainability factors identified, eight were related to current code smell definitions. This provides an insight on the potential of code smells to cover maintainability factors and to complement the analyses based on software metrics and expert judgment. In some cases, code smells would need to be complemented with alternative approaches, such as expert judgment (see Refs. [6, 61]) and semantic analysis techniques (for example, see Maletic et al. [83]) to achieve a comprehensive assessment of maintainability.

4.6 A Technique for Integrating Code Smell Analysis and Expert Judgment

A technique originally from the domain of social research called concept mapping [143] was adapted to a software engineering context. We used this technique to guide the selection, the analysis, and the interpretation of code smells via expert judgment.

We compared the maintainability maps[2] of the four systems using the concept mapping technique with the expert-judgment-based assessment reported in Ref. [6]. The maintainability constructs found using concept mapping were similar to the aspects reported by the experts in Ref. [6], and the values of those constructs for the four systems were in

[2]Maintainability map consists of a pictorial representation of the measurements of the different code smells operationalizing different maintenance constructs.

accordance with the experts' assessments of the differences between the systems in terms of maintainability.

1. Concept mapping provides a technique for explicit descriptions of the *criteria* and the *process* used for mapping code smells (or any measurable attribute of the code) to high-level, abstract quality constructs. This seems to be a useful addition to alternative approaches, such as the Goal-Question-Metric and the Factor Criteria Metric approaches. Concept mapping also facilitates the understanding of the rationale behind the operationalization of the quality constructs. Also, it may consequently simplify the interpretation of the measures behind the constructs.

2. Concept mapping provides a pictorial representation where the measures operationalizing a construct are explicit. This allows the observation of the measures in relation to all other measures and enables us to see how each of them fits into the overall picture. In our case, the multivariate representations supported the comparison of data across systems, as well as the interpretation of data.

3. Concept mapping can be used to generate representations of maintainability according to the context of the assessment. This enables a structured process for considering contextual aspects, such as cost, urgency, and difficulty of a maintenance task. This may also guide the process of defining the most desirable characteristics in a system for particular maintenance contexts.

4. Although concept mapping provides a structured set of steps, the technique may be considered rather complex and time-consuming in industry settings. As such, a cost-benefit analysis should always be considered, with the complexity of the problem to be addressed and the cost of conducting concept mapping as input.

5. It is not clear which of the visualization techniques could provide the best insight for code smell analysis at the system level nor which aggregation operations (sum, average, density, etc.) would result in the best overview of the different maintainability constructs. Also, it is not obvious which "perspectives" (i.e., risk management or adherence to OO principles) are best suited to represent a given assessment context. This lack of clarity points at the importance of skills and experience in the underlying technique and domains to be able to use concept mapping efficiently.

4.7 Implications and Recommendations

This section provides a list of implications and recommendations derived from the analyses presented in the six papers.

The results in Paper 1 indicate that the correspondence between code-smell-based system-level measures and maintenance outcome measures depends on whether the systems are of similar sizes or not. When systems of similar sizes are compared, we may use code smell density. Otherwise, we may benefit from using the number of code smells to indicate differences in maintainability, as measured by maintenance effort and defects. Our analysis comparing different system-level maintainability assessment approaches indicates that such assessments are likely to benefit from the use of expert judgment. Expert judgment, given a good expert, provides a higher chance of adapting to different maintenance contexts, such as the differences related to the complexity of the tasks, the size of the systems, and the maintenance goals. We do, however, believe that expert judgment and code smells complement each other because code-smell-based analyses may sometimes spot potentially problematic areas of the system missed by the expert. Consequently, we recommend the use of code smells in combination with expert judgment to support more comprehensive and accurate system-level assessments of maintainability.

The results in Paper 2 suggest that *file size* and *number of revisions* are stronger indicators of effort to maintain a file than the presence of code smells. This implies that to reduce the reading and updating effort of a file during maintenance work, it is more important to strive for the reduction of a code rather than the removal of any specific code smell. Also, the presence of the code smell Refused Bequest indicates a significantly less reading and updating effort of that file. A similar result was found for the code smell Data Clump in the study reported in Paper 3. This suggests that in some contexts, there may be potentially *positive* effects from the presence of some code smells. Based on this result, we recommend that at least these two code smells be investigated further to determine if the underlying concept of the code smell has, in fact, positive effects on maintainability or if the detection strategy used for the code smell leads to false negatives (i.e., investigate whether there is a lack of correspondence between the original concepts of the code smells and their detection implementations).

The results in Paper 3 indicate that the ISP Violation in a file leads to a higher likelihood of that file being problematic during maintenance. An exploratory factor analysis indicates that ISP Violation and Shotgun Surgery tend to occur in the same file and are rather independent of the factor related to the size of a file. These two code smells are *size-independent* indicators of problematic codes. We recommend examining for their presence in the files to identify potentially problematic areas and to prioritize refactoring efforts. Paper 3 also reports that the study of interaction effects between code smells and between a code smell and other design flaws should be investigated further to understand better the effects of code smells. This is similar to the notion of inter-smell relations, as proposed by Walter and Pietrzak [148]. A potentially meaningful approach for analyzing

the interactions between code smells is to observe and analyze the effects of code smells collocated in the same file on maintainability by using the exploratory factor analysis we applied in Paper 3.

Paper 4 reports that only about 30% of the total amount of maintenance problems could potentially (best case) be explained and predicted by the presence of code smells. This means that aspects covered by current code smell detection methods may have a relatively low potential in explaining and predicting outcomes of a maintenance project. These results may, to some extent, explain the findings reported in Papers 2 and 3 (i.e., the lack of significant coefficients for most of the code-smell-related variables in the regression analyses). This supports the recommendation that code smell analysis should be done in combination with alternative maintainability assessment strategies, such as expert-judgment-based assessments.

Additional observations led us to believe that we should have only a modest expectation of the explanatory and predictive power of *individual* code smells in relation to software maintainability. Paper 4 reports that interaction effects occur between code smells and between code smells and other design flaws. This implies that the current approach for code smell analysis (i.e., analyzing individual smells and not the effect of their combinations) limits the capability of code smells to explain much of the maintenance problems caused by design flaws. Another limitation of the current approaches for code smell analysis is that couplings among elements containing code smells (e.g., files containing code smells) are not considered in the analyses. The findings from Papers 3 and 4 indicate that interaction effects between code smells distributed across coupled files may have the same consequences from a practical perspective as interaction effects between code smells collocated in the same file. This last finding implies a serious consideration for further studies on code smells and a need to include dependency analyses to provide a better understanding of the role of code smells in software maintenance.

In the analysis in Paper 5, many maintainability factors deemed critical not only by the software experts in Ref. [6], who evaluated the maintainability of the four systems, but also by the developers who maintained the systems, were not addressable via code smell analysis. This result supports the recommendation that alternative approaches should be combined with code smell analysis to achieve better assessments of maintainability. Based on the maintainers' assessments, the development and improvement of detection strategies that focus on code smells related to the factor "design consistency" should be prioritized, perhaps including the development of new code smells.

Results from Paper 6 imply that there are proven methods/techniques outside the software engineering domain that can be used to address some of the challenges and limitations entailed by maintainability assessments based on code smells. We propose the

use of concept mapping as a promising technique for this purpose.

4.8 Limitations of the Work

This section first addresses the limitations of the empirical study used as the basis for the papers included in this thesis and then presents the specific limitations of the individual papers.

4.8.1 Limitations of the Study Design

***Construct validity*:** With respect to the code smell construct, we used automated detection to avoid subjective bias. Automated detection may have false negatives, and the meaningfulness and/or lack of standard detection strategies used in the tools could be a potential threat. We are aware that there are other tools that can detect many of the code smells we analyzed and that their detection strategies could differ to an extent from those used in the study. In spite of this, we believe that our choice of tools, with commonly accepted implementations of detection strategies for the code smells, limits the construct validity threat of our study to some extent.

***Internal validity*:** Several moderator variables were controlled to reduce the threats to internal validity. For instance, the fact that all four analyzed systems had nearly identical functionalities removes some of the internal validity problems that often appear when comparing different systems in noncontrolled environments. Special attention was given to ensure a similar environment and development technology across the maintenance tasks and systems. Within the maintenance project, particular effort was spent on recruiting developers with nearly similar skills by using a skill instrument reported in a previously published peer-reviewed work.

***External validity*:** The results from this study should mainly be interpreted within the context of medium sized, Java-based, web information systems and medium to small maintenance tasks. The programmers completed the tasks individually, i.e., not by teams or by using pair programming. This last characteristic can affect the applicability of the results in highly collaborative environments. The average efforts were approximately 26 hours for tasks 1 and 2 and approximately 6 hours for task 2. Despite the fact that there is still no well-defined classifications of the size of maintenance tasks in software engineering [133], we believe that tasks 1 and 3 are medium-sized maintenance tasks and task 2 is a small maintenance task. The tasks represent typical maintenance scenarios and are based on real maintenance needs.

This work does not claim the results to fully represent long-term maintenance projects with large tasks, given the smaller size of the tasks and the shorter maintenance period covered in our study. The tasks involved may better resemble backlog items in a single sprint or iteration within the context of an *Agile* development.

***Repeatability*:** Case studies are typically difficult to repeat due to their rich context, which can hardly be described in full detail. This may, to some extent, be a problem in our study as well. However, we have kept a comprehensive documentation of the study protocol and the procedures for conducting the interviews, for the think-aloud sessions, and for summarizing and analyzing the data. This documentation will be sent to interested researchers upon request to the author of this thesis. The existence of comprehensive documentation means that we have, to some extent, ensured that other studies may repeat our study in a similar context and compare their results with our findings.

4.8.2 Limitations Specific to Each Paper

Paper 1. In this paper, maintainability (maintenance outcome) was defined through the measures–effort and number of defects introduced. For each of these measures, data were collected from several sources, which allowed triangulation[3]. A potential threat in the design of the study was the difference in effort between "rounds" (i.e., a system being maintained for the first time or when the developers already have completed the tasks in a previous system). To eliminate this threat, only effort and defects from first rounds were included in the analysis.

Paper 2. This work analyzed the relationship between the amount of effort spent on maintaining a file and the presence of code smells in that file. However, developers may work around smelly files (i.e., instead of modifying a smelly file, developers could find it easier to duplicate code fragments of the file). If a piece of code is copied into a new file and the modified functionality is implemented there, the effort of the modification would be associated with the new file instead of the smelly one. To investigate this potential threat, we identified the code that was copied across files in the four systems by using the Simian tool [52], and we found that the probability of such copying was independent of the presence of code smells in the original file.

[3]In the social sciences, triangulation is often used to indicate that more than two methods are used in a study, with a view to double (or triple) checking results. This is also called "cross examination."

Papers 3 and 4. To measure maintainability, instances of *maintenance problems* were identified and collected through interviews and think-aloud sessions. It is possible that some developers were more open than others and reported more maintenance problems and that some developers did not report all the maintenance problems they experienced. The use of three independent collection methods, i.e., interviews, direct observation, and think-aloud sessions, for triangulation purposes may have reduced this threat. A limitation, particularly in Paper 3, is that the severity level of a maintenance problem was not collected or assessed.

As with most other qualitative research, researcher bias may occur when selecting data to analyze and report and when summarizing and interpreting the data. To reduce this threat, the set of qualitative data was partially assessed by a second researcher to verify the interpretations of the researcher who performed the qualitative analysis. If there were disagreements on a given interpretation, a re-examination of different data sources was done, followed by a discussion between the researchers.

Paper 5. Following a *grounded theory* perspective, instead of addressing the typical threats to validity in empirical studies, we recommend the evaluation of the quality aspects of the findings, such as their fit[4] and relevance[5] [46]. From these perspectives, we argue that the software maintainability factors identified by the experts who assessed the systems corresponded to the factors mentioned by the developers who maintained the systems, and in most of the cases, both parts agreed on the relevance of those factors.

Paper 6. The major limitation in this paper is the fact that the participants used as experts (except for one software engineer) in the concept mapping session constituted a convenience sample. Thus, they may not necessarily behave like software professionals would do in industrial settings. Because of the exploratory nature of this study, the intended contribution may be more dependent on the description of the experience with the applied concept mapping technique than on the direct results from its application. In spite of this limitation, given that this technique is relatively unknown in software engineering, we believe that a detailed account of how to apply it on nontrivial software engineering problems may help in the improvement of current practices.

[4]This has to do with how closely concepts fit with the incidents they are representing, and this is related to how thoroughly the constant comparison of incidents to concepts was done.

[5]A relevant study deals with the real concerns of participants (captures their attention) and is not only of academic interest.

4.9 Directions for Future Research

The areas for future work identified through this research include the following:

Interaction effects among code smells: More focus is needed on the implications of combinations of code smells (and other types of design flaws) on maintainability instead of investigating only the effects of individual code smells (this corresponds with the ideas of Walter and Pietrzak [148]). This entails building more comprehensive symptomatic characterizations of different types of potential maintenance problems (e.g., in the form of *inter-smell relations*) and uncovering the causal mechanisms that lead to them.

Study of collocated smells and coupled smells: More focus is needed on dependency analysis alongside the analysis of interaction effects across code smells. We suggest this among others because interactions between code smells can occur across coupled files. This interaction is currently ignored due to the fact that code smells are mostly analyzed at the file level and "coupled code smells" are not identified. Also, the dependency analysis should focus on *types* of dependencies (e.g., data, functional, abstract definition, and inheritance) and their *quantifiable attributes* (e.g., intensity, spread, and depth).

Nature and severity of maintenance problems: Future work should focus on quantifying the severity and the degree of the impact of different types of maintenance problems in different contexts to establish the *relative* importance and context dependency of code smells. This way, it may be possible to assess not only whether and how code smells cause maintenance problems, but also *how much* those problems matter on concrete outcomes of maintenance projects compared with other problems and in different contexts.

Cost-/benefit-based definition/detection of code smells: Further research should focus on defining and extending a catalog of design factors that have empirical evidence of their relevance on maintainability. This catalog may be used to guide further efforts in new definitions of code smells and corresponding detection methods/tools.

Further refinement and evaluation of concept mapping: Further research on the use of concept mapping in industry-relevant assessment contexts (and its comparison with other similar techniques) should be completed to provide better insight into the adequacy and suitability of this technique.

Chapter 5

Concluding Remarks

It has been, and still is, a major challenge to evaluate software quality characteristics such as maintainability through software measures and indicators in realistic software development contexts. Research from the past decade has emphasized the formalization and automated detection of code smells for this purpose, but little has been done to investigate how comprehensive and informative these indicators actually are for the purpose of assessing maintainability in realistic software development situations. The research presented in this thesis empirically investigates the actual *applicability* of such code smells (including capabilities and limitations) for conducting software maintainability assessments in a relevant and realistic context of software maintenance.

One goal of the research was to enable an evaluation of the usefulness of code smells for system-level maintainability assessments. We found that many code smells are correlated with size. Therefore, aggregation of code smells at the system level may not add much useful information for maintainability comparisons not already existing when comparing the systems sizes. Also, the use of code smell density (which enables us to adjust for size differences to some extent) is advisable only when the maintainability of systems with similar sizes is compared.

Another goal was to better understand the capability of code smells to locate and explain differences in software maintainability, measured as maintenance effort to update a file or problems related to a file. The regression-model-based analyses indicate that only one code smell, i.e., ISP Violation, contributed significantly to the explanation of lower maintainability. Two code smells, i.e., Data Clump and Refused Bequest, even had positive effects on maintainability.

This thesis includes different perspectives of maintainability to better understand the role that code smells play in maintenance. Thus, outcome-based interpretations of maintainability (e.g., effort, defects, and change size) were considered. The more qualitative, process-related aspects (e.g., number and types of maintenance problems and develop-

ers' perception of maintainability) were investigated as well. Results from this *multiple-perspective analysis* indicate, for instance, that code smells are better suited to locate codes that are likely to be perceived as problematic by developers (i.e., ISP Violation) than to explain maintenance outcome measures, such as effort.

Our examination of maintenance problems encountered by developers uncovered several limitations of code smells when used to assess the overall maintainability of a system and demonstrated that alternative methods are required to address those limitations. Results from the qualitative analysis demonstrated how current approaches for code-smell-based analysis may miss key aspects (e.g., the interaction effects between code smells) that may sometimes be essential to explain the relation between code smells and maintenance problems.

This research also demonstrates that the incorporation of contextual information pertaining to maintenance (e.g., size and type of maintenance task) is essential to give an accurate and useful interpretation of measures describing software design. In accordance to this finding, a methodological contribution (adaptation of the concept mapping technique for software maintainability assessment purposes) is provided for building, adapting, and validating such quality models for a given context or setting.

References for the Summary

[1] Marwen Abbes et al. "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension." In: *European Conf. Softw. Maint. and Reeng.* 2011, pp. 181–190.

[2] Alain Abran and Hong Nguyenkim. "Analysis of maintenance work categories through measurement." In: *Int'l Conf. Softw. Maint.* 1991, pp. 104–113.

[3] Yunsik Ahn et al. "The software maintenance project effort estimation model based on function points." In: *Journal of Software Maintenance* 15.2 (2003), pp. 71–85.

[4] El Hachemi Alikacem and Houari A. Sahraoui. "A Metric Extraction Framework Based on a High-Level Description Language." In: *Working Conf. Source Code Analysis and Manipulation.* 2009, pp. 159–167.

[5] Mohammed Alshayeb and Li Wei. "An empirical validation of object-oriented metrics in two different iterative software processes." In: *IEEE Transactions on Software Engineering* 29.11 (2003), pp. 1043–1049.

[6] Bente C. D. Anda. "Assessing Software System Maintainability using Structural Measures and Expert Assessments." In: *Int'l Conf. Softw. Maint.* 2007, pp. 204–213.

[7] Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System." In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 407–429.

[8] Carina Andersson and Per Runeson. "A spiral process model for case studies on software quality monitoring method and metrics." In: *Software Process: Improvement and Practice* 12.2 (2007), pp. 125–140.

[9] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. "How Clones are Maintained: An Empirical Study." In: *European Conf. Softw. Maint. and Reeng.* 2007, pp. 81–90.

[10] Tibor Bakota. “Tracking the evolution of code clones.” In: *Int’l Conf. on Current trends in theory and practice of computer science*. 2011, pp. 86–98.

[11] Rajiv D. Banker, B. Davis Gordon, and Sandra A. Slaughter. “Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study.” In: *Management Science* 44.4 (1998), pp. 433–450.

[12] Izak Benbasat, David K. Goldstein, and Melissa Mead. “The Case Research Strategy in Studies of Information Systems.” In: *MIS Quarterly* 11.3 (1987), pp. 369–386.

[13] Hans Benestad, Bente Anda, and Erik Arisholm. “Assessing Software Product Maintainability Based on Class-Level Structural Measures.” In: *Product-Focused Softw. Process Improvement*. 2006, pp. 94–111.

[14] Keith H. Bennett. “An introduction to software maintenance.” In: *Journal of Information and Software Technology* 12.4 (1990), pp. 257–264.

[15] Gunnar R. Bergersen and Jan-Eric Gustafsson. “Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective.” In: *Journal of Individual Differences* 32.4 (2011), pp. 201–209.

[16] Gunnar R. Bergersen et al. “Inferring Skill from Tests of Programming Performance: Combining Time and Quality.” In: *Int’l Conf. Softw. Eng. and Measurement*. 2011, pp. 305–314.

[17] Borland. *Borland Together [online] Available at: http://www.borland.com/us/products/together [Accessed 10 May 2012]*. 2012.

[18] Mark G. J. Van den Brand et al. “Using The Meta-Environment for Maintenance and Renovation.” In: *European Conf. Softw. Maint. and Reeng.* 2007, pp. 331–332.

[19] William Brown et al. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[20] Sergio Bryton, Fernando Brito e Abreu, and Miguel Monteiro. “Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method.” In: *Int’l Conf. Quality of Inf. and Comm. Techn.* 2010, pp. 337–342.

[21] Glauco De F. Carneiro et al. “Identifying Code Smells with Multiple Concern Views.” In: *Brazilian Symposium on Software Engineering*. 2010, pp. 128–137.

[22] Alexander Chatzigeorgiou and Anastasios Manakos. “Investigating the Evolution of Bad Smells in Object-Oriented Code.” In: *Int’l Conf. Quality of Inf. and Comm. Techn.* 2010, pp. 106–115.

[23] Ram Chillarege et al. “Orthogonal Defect Classification-A Concept for In-Process Measurements.” In: *IEEE Transactions on Software Engineering* 18 (11 1992), pp. 943–956.

[24] Peter Coad and Edward Yourdon. *Object-Oriented Design. London,* London: Prentice Hall, 1991.

[25] Vianney Cote and Denis St.Pierre. “A model for estimating perfective software maintenance projects.” In: *Int’l Conf. Softw. Maint.* 1990, pp. 328–334.

[26] Steve Counsell. “Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?” In: *Int’l Conf. on Research Challenges in Information Science*. 2008, pp. 111–122.

[27] Cummings and Worley. *Organization Development and Change*. Thomson, 2005.

[28] Marco D’Ambros, Alberto Bacchelli, and Michele Lanza. “On the Impact of Design Flaws on Software Defects.” In: *Int’l Conf. Quality Softw.* 2010, pp. 23–31.

[29] Andrea De Lucia, Eugenio Pompella, and Silvio Stefanucci. “Assessing effort estimation models for corrective maintenance through empirical studies.” In: *Inf. and Softw. Tech.* 47.1 (2005), pp. 3–15.

[30] Ignatios Deligiannis et al. “A controlled experiment investigation of an object-oriented design heuristic for maintainability.” In: *Journal of Systems and Software* 72.2 (2004), pp. 129–143.

[31] Ignatios Deligiannis et al. “An empirical investigation of an object-oriented design heuristic for maintainability.” In: *Journal of Systems and Software* 65.2 (2003), pp. 127–139.

[32] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. “Visual Detection of Design Anomalies.” In: *European Conf. Softw. Maint. and Reeng.* 2008, pp. 279–283.

[33] Edgewall-Software. *Trac [online] Available at: http://trac.edgewall.org [Accessed 10 May 2012]*. 2012.

[34] Elaine H. Ferneley. “Design metrics as an aid to software maintenance: an empirical study.” In: *Journal of Software Maintenance* 11.1 (1999), pp. 55–72.

[35] Fabrizio Fioravanti. “Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems.” In: *IEEE Transactions on Software Engineering* 27 (2001), pp. 1062–1084.

[36] Beat Fluri. “Assessing Changeability by Investigating the Propagation of Change Types.” In: *Int’l Conf. on Softw. Eng.* 2007, pp. 97–98.

[37] Bent Flyvberg. *Five misunderstandings about case-study research.* In Qualitative Research Practice. Sage, 2007.

[38] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of feature envy bad smells." In: *Int'l Conf. Softw. Maint.* 2007, pp. 519–520.

[39] Francesca Arcelli Fontana et al. "An Experience Report on Using Code Smells Detection Tools." In: *Int'l Conf. on Softw. Testing, Verification and Validation.* 2011, pp. 450–457.

[40] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[41] Alfonso Fuggetta et al. "Applying GQM in an industrial software factory." In: *ACM Trans. Softw. Eng. Meth.* 7.4 (1998), pp. 411–448.

[42] Erich Gamma et al. *Design Patterns. Elements of Reusable Software.* Addison-Wesley, 1995.

[43] Reto Geiger et al. "Relation of code clones and change couplings." In: *Fundamental Approaches to Software Engineering, Proceedings.* Vol. 3922. 2006, pp. 411–425.

[44] Genuitec. *My Eclipse [online] Available at: http://www.myeclipseide.com [Accessed 10 May 2012].* 2012.

[45] Alexander L. George and Andrew Bennett. *Case studies and theory development in the social sciences.* Cambridge, Massachusetts: MIT Press, 2005.

[46] Barney G. Glaser and Anselm L. Strauss. *The discovery of grounded theory: Strategies for qualitative research.* Aldine Pub. Co, 1967.

[47] Nils Göde and Jan Harder. "Oops! . . . I changed it again." In: *Int'l Workshop on Software Clones.* 2011, pp. 14–20.

[48] Juan Carlos Granja-Alvarez and Manuel José Barranco-García. "A Method for Estimating Maintenance Cost in a Software Project." In: *Journal of Software Maintenance* 9.3 (1997), pp. 161–175.

[49] Gregory A. Hall and John C. Munson. "Software evolution: code delta and code churn." In: *Journal of Systems and Software* 54.2 (2000), pp. 111–118.

[50] Jo E. Hannay, Dag I. K. Sjøberg, and Tore Dybå. "A Systematic Review of Theory Use in Software Engineering Experiments." In: *IEEE Transactions on Software Engineering* 33.2 (2007), pp. 87–107.

[51] Haralambi Haralambiev et al. "Applying source code analysis techniques." In: *Int'l Conf. on Computer as a Tool.* 2011, pp. 2–3.

[52] Simon Harris. *Simian [online] Available at: http://www.harukizaemon.com/simian/index.html [Accessed 5 January 2012]*. 2011.

[53] Warren Harrison and Curtis Cook. "Insights on improving the maintenance process through software measurement." In: *Int'l Conf. Softw. Maint.* 1990, pp. 37–45.

[54] Salima Hassaine et al. "IDS: An Immune-Inspired Approach for the Detection of Software Design Smells." In: *Int'l Conf. Quality of Inf. and Comm. Techn.* Ieee, Sept. 2010, pp. 343–348.

[55] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability." In: *Int'l Conf. Quality of Inf. and Comm. Techn.* 2007, pp. 30–39.

[56] Steffen Herbold, Jens Grabowski, and Helmut Neukirchen. "Automated Refactoring Suggestions Using the Results of Code Analysis Tools." In: *Int'l Conf. on Advances in System Testing and Validation Lifecycle*. 2009, pp. 104–109.

[57] Allyson Hoss and Doris L. Carver. "An ontological identification of relationships between anti-patterns and code smells." In: *IEEE Aerospace Conference*. Mar. 2010, pp. 1–10.

[58] Intooitus. *InCode [online] Available at: http://www.intooitus.com/inCode.html [Accessed 10 May 2012]*. 2012.

[59] ISO/IEC. *International Standard ISO/IEC 9126*. Tech. rep. Geneva: International Organization for Standardization, 1991.

[60] ISO/IEC. *ISO/IEC Technical Report 19759:2005*. 2005.

[61] Magne Jørgensen. "Estimation of Software Development Work Effort:Evidencence on Expert Judgment and Formal Models." In: *International Journal of Forecasting* 23.3 (2007), pp. 449–462.

[62] Padmaja Joshi and Rushikesh K. Joshi. "Concept Analysis for Class Cohesion." In: *European Conf. Softw. Maint. and Reeng.* 2009, pp. 237–240.

[63] Elmar Juergens et al. "Do code clones matter?" In: *Int'l Conf. Softw. Eng.* 2009, pp. 485–495.

[64] Mira Kajko-Mattsso et al. "A Model of Maintainability - Suggestion for Future Research." In: *Software Engineering Research and Practice*. CSREA Press, 2006, pp. 436–441.

[65] Cory Kapser and Michael Godfrey. "'Cloning considered harmful' considered harmful: patterns of cloning in software." In: *Empirical Software Engineering* 13 (6 2008), pp. 645–692.

[66] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. “An Exploratory Study of the Impact of Code Smells on Software Change-proneness.” In: *Working Conf. Reverse Eng.* 2009, pp. 75–84.

[67] Gregor Kiczales et al. “Open implementation design guidelines.” In: *Int’l Conf. on Softw. Eng.* 1997, pp. 481–490.

[68] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. “Mining Software Repositories with iSPAROL and a Software Evolution Ontology.” In: *Int’l Workshop on Mining Software Repositories*. 2007, p. 10.

[69] Miryung Kim et al. “An empirical study of code clone genealogies.” In: *European Softw. Eng. Conf. and ACM SIGSOFT Symposium on Foundations of Softw. Eng.* 2005, pp. 187–196.

[70] Barbara A. Kitchenham et al. “Towards an ontology of software maintenance.” In: *Journal of Software Maintenance* 11.6 (1999), pp. 365–389.

[71] Jussi Koskinen and Tero Tilus. *Software maintenance cost estimation and modernization support*. Tech. rep. Information Technology Research Institute, University of Jyvaskyla, 2003, p. 62.

[72] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2005, p. 220.

[73] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2004.

[74] Lucas M. Layman, Laurie A. Williams, and Robert St. Amant. “MimEc.” In: *Proceedings of the 2008 Int’l workshop on cooperative and human aspects of softw. engin.* 2008, pp. 73–76.

[75] Meir M. Lehman, D. E. Perry, and Juan F. Ramil. “Implications of Evolution Metrics on Software Maintenance.” In: *Int’l Conf. on Softw. Maint.* 1998, pp. 208–.

[76] Hareton K. N. Leung. “Estimating Maintenance Effort by Analogy.” In: *Empirical Software Engineering* 7.2 (2002), pp. 157–175.

[77] Wei Li and Raed Shatnawi. “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.” In: *Journal of Systems and Software* 80.7 (2007), pp. 1120–1128.

[78] Karl J. Lieberherr and Ian M. Holland. “Assuring Good Style for Object-Oriented Programs.” In: *IEEE Software* 6.5 (1989), pp. 38–48.

[79] Hui Liu et al. “Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort.” In: *IEEE Transactions on Software Engineering* 38.1 (Jan. 2012), pp. 220–235.

[80] Angela Lozano and Michel Wermelinger. “Assessing the effect of clones on changeability.” In: *Int'l Conf. Softw. Maint.* 2008, pp. 227–236.

[81] Isela Macia, Alessandro Garcia, and Arndt Von Staa. “Defining and Applying Detection Strategies for Aspect-Oriented Code Smells.” In: *Brazilian Symposium on Software Engineering*. Sept. 2010, pp. 60–69.

[82] Isela Macia et al. “On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study.” In: *Brazilian Symposium on Software Components, Architectures and Reuse*. 2011, pp. 41–50.

[83] Jonathan I. Maletic and Andrian Marcus. “Supporting program comprehension using semantic and structural information.” In: *Int'l Conf. Softw. Eng.* 2001, pp. 103–112.

[84] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. “A taxonomy and an initial empirical study of bad smells in code.” In: *Int'l Conf. Softw. Maint.* 2003, pp. 381–384.

[85] Mika V. Mäntylä. “An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement.” In: *Int'l Conf. Softw. Eng.* 2005, pp. 277–286.

[86] Leandra Mara et al. “Hist-Inspect: a tool for history-sensitive detection of code smells.” In: *Int'l Conf. on Aspect-oriented software development companion*. 2011, pp. 65–66.

[87] Radu Marinescu. “Measurement and Quality in Object Oriented Design.” Doctoral Thesis. “Politehnica" University of Timisoara, 2002.

[88] Radu Marinescu. “Measurement and quality in object-oriented design.” In: *Int'l Conf. Softw. Maint.* 2005, pp. 701–704.

[89] Radu Marinescu and Daniel Ratiu. “Quantifying the quality of object-oriented design: the factor-strategy model.” In: *Working Conf. Reverse Eng.* 2004, pp. 192–201.

[90] Robert C. Martin. *Agile Software Development, Principles, Patterns and Practice*. Prentice Hall, 2002.

[91] Karl S. Mathias et al. “The role of software measures and metrics in studies of program comprehension.” In: *ACM Southeast regional Conf.* 1999, p. 13.

[92] Jean Mayrand and Francois Coallier. “System acquisition based on software product assessment.” In: *Int’l Conf. Softw. Eng.* 1996, pp. 210–219.

[93] Tom Mens, Tom Tourwe, and Francisca Munoz. “Beyond the refactoring browser: advanced tool support for software refactoring.” In: *Int’l Workshop on Principles of Software Evolution*. 2003, pp. 39–44.

[94] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall PTR, 1997.

[95] Naoel Moha, Yann-Gaël Guéhéneuc, and Anne-Francoise Le Meur. “From a domain analysis to the specification and detection of code and design smells.” In: *Formal Aspects of Computing*. Vol. 22. Springer, 2010, pp. 345–361.

[96] Naouel Moha. “Detection and correction of design defects in object-oriented designs.” In: *ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications*. 2007, pp. 949–950.

[97] Naouel Moha, Yann-Gael Gueheneuc, and Pierre Leduc. “Automatic generation of detection algorithms for design defects.” In: *IEEE/ACM Int’l Conf. on Automated Softw. Eng.* 2006, pp. 297–300.

[98] Naouel Moha et al. “A domain analysis to specify design defects and generate detection algorithms.” In: *Fundamental Approaches to Softw. Eng.* 4961 (2008), pp. 276–291.

[99] Naouel Moha et al. “DECOR: A Method for the Specification and Detection of Code and Design Smells.” In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[100] Akito Monden et al. “Software quality analysis by code clones in industrial legacy software.” In: *IEEE Symposium on Software Metrics*. 2002, pp. 87–94.

[101] Miguel P. Monteiro and Joao M. Fernandes. “Towards a catalog of aspect-oriented refactorings.” In: *Int’l Conf. on Aspect-oriented software development*. 2005, pp. 111–122.

[102] Matthew James Munro. “Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code.” In: *Int’l Software Metrics Symposium*. 2005, pp. 15–.

[103] Emerson Murphy-Hill. “Scalable, expressive, and context-sensitive code smell display.” In: *ACM SIGPLAN Conf. on Object-oriented programming systems languages and applications*. 2008, pp. 771–772.

[104] S. Muthanna et al. “A maintainability model for industrial software systems using design level metrics.” In: *Working Conf. Reverse Eng.* 2000, pp. 248–256.

[105] Nachiappan Nagappan and Thomas Ball. “Use of relative code churn measures to predict system defect density.” In: *Int’l Conf. Softw. Eng.* 2005, pp. 284–292.

[106] Niclas Ohlsson. “Predicting Fault-Prone Software Modules in Telephone Switches.” In: *IEEE Transactions on Software Engineering* 22 (1996), pp. 886–894.

[107] Steffen Olbrich et al. “The evolution and impact of code smells: A case study of two open source systems.” In: *Int’l Symposium on Empirical Softw. Eng. and Measurement*. 2009, pp. 390–400.

[108] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. “Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems.” In: *Int’l Conf. Softw. Maint.* 2010, pp. 1–10.

[109] Paul Oman and Jack Hagemeister. “Construction and testing of polynomials predicting software maintainability.” In: *Journal of Systems and Software* 24.3 (1994), pp. 251–266.

[110] Paul Oman and Jack Hagemeister. “Metrics for assessing a software system’s maintainability.” In: *Int’l Conf. Softw. Maint.* 1992, pp. 337–344.

[111] Oracle. *My Sql [online] Available at: http://www.mysql.com [Accessed 10 May 2012]*. 2012.

[112] David L. Parnas. “On the criteria to be used in decomposing systems into modules.” In: *Communications of ACM* 15.12 (1972), pp. 1053–1058.

[113] Chris Parnin, Carsten Görg, and Ogechi Nnadi. “A catalogue of lightweight visualizations to support code smell inspection.” In: *ACM Symposium on Software visualization*. 2008, pp. 77–86.

[114] Ralph Peters and Andy Zaidman. “Evaluating the Lifespan of Code Smells using Software Repository Mining.” In: *European Conf. on Softw. Maint. and Reeng.* 2012, pp. 411–416.

[115] Blazej Pietrzak and Bartosz Walter. “Leveraging code smell detection with inter-smell relations.” In: *Int’l Conf. on Extreme Programming and Agile Processes in Software Engineering*. 2006, pp. 75–84.

[116] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, 1996, p. 384.

[117] Eduardo Piveta et al. “Representing refactoring opportunities.” In: *Proceedings of the 2009 ACM symposium on Applied Computing*. 2009, pp. 1867–1872.

[118] Markus Pizka and Florian Deissenboeck. “How to effectively define and measure maintainability.” In: *Softw. Measurement European Forum*. 2007.

[119] Plone Foundation. *Plone CMS: Open Source Content Management [online] Available at: http://plone.org [Accessed 10 May 2012]*. 2012.

[120] Charles Ragin. *The comparative method: Moving beyond qualitative and quantitative strategies*. Berkeley: University of California Press, 1987.

[121] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. "Clones: What is that smell?" In: *Int'l Conf. Softw. Eng.* 2010, pp. 72–81.

[122] Juan F. Ramil and Meir M. Lehman. "Metrics of software evolution as effort predictors – A case study." In: *Int'l Conf. Softw. Maint.* 2000, pp. 163–172.

[123] Ananda Rao and Narendar Reddy. "Detecting bad smells in object oriented design using design change propagation probability matrix." In: *Int'l Multiconference of Engineers and Computer Scientists*. 2008, pp. 1001–1007.

[124] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. "A systematic review of software maintainability prediction and metrics." In: *Int'l Conf. Softw. Eng. and Measurement*. 2009, pp. 367–377.

[125] Arthur J. Riel. *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley, 1996.

[126] Colin Robson. *Real World Research*. 2nd. Blackwell Publishing, 2002.

[127] Tony Rosqvist, Mika Koskela, and Hannu Harju. "Software Quality Evaluation Based on Expert Judgement." In: *Softw. Quality Control* 11.1 (2003), pp. 39–55.

[128] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering." In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164.

[129] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws." In: *Int'l Conf. on Program Comprehension*. 2006, pp. 159–168.

[130] Outi Salo and Peka Abrahamsson. "Empirical Evaluation of Agile Software Development: The Controlled Case Study Approach." In: *Product-Focused Softw. Process Improvement*. 2004, pp. 408–423.

[131] Olaf Seng, Johannes Stammel, and David Burkhart. "Search-based determination of refactorings for improving the class structure of object-oriented systems." In: *Annual conference on Genetic and evolutionary computation*. 2006, pp. 1909–1916.

[132] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. "Metrics Based Refactoring." In: *European Conf. Softw. Maint. and Reeng.* 2001, pp. 30–.

[133] Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. "The Future of Empirical Methods in Software Engineering Research." In: *Future of Software Engineering (FOSE)* SE-13.1325 (2007), pp. 358–378.

[134] K. Srivisut and P. Muenchaisri. "Bad-Smell Metrics for Aspect-Oriented Software." In: *IEEE/ACIS Int'l Conf. on Computer and Information Science*. 2007, pp. 1060–1065.

[135] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[136] Giancarlo Succi et al. "Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics." In: *Journal of Systems and Software* 65.1 (2003), pp. 1–12.

[137] The Apache Software Foundation. *Apache Subversion [online] Available at: http://subversion.apache.org [Accessed 10 May 2012]*. 2012.

[138] The Apache Software Foundation. *Apache Tomcat [online] Available at: http://tomcat.apache.org [Accessed 10 May 2012]*. 2012.

[139] TMate-Sofware. *SVNKit - Subversioning for Java. [online] Available at: http://svnkit.com [Accessed 10 May 2012]*. 2010.

[140] Tom Tourwé and Tom Mens. "Identifying Refactoring Opportunities Using Logic Meta Programming." In: *European Conf. Softw. Maint. and Reeng.* 2003, p. 91.

[141] Guilherme Travassos et al. "Detecting defects in object-oriented designs." In: *ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*. 1999, pp. 47–56.

[142] Adrian Trifu and Urs Reupke. "Towards Automated Restructuring of Object Oriented Systems." In: *European Conf. Softw. Maint. and Reeng.* 2007, pp. 39–48.

[143] William M. K. Trochim. "An introduction to concept mapping for planning and evaluation." In: *Evaluation and program planning* 12.1 (1989), pp. 1–16.

[144] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of type-checking bad smells." In: *European Conf. Softw. Maint. and Reeng.* 2008, pp. 329–331.

[145] Nikolaos Tsantalis and Alexander Chatzigeorgiou. "Ranking Refactoring Suggestions Based on Historical Volatility." In: *European Conf. Softw. Maint. and Reeng.* Ieee, Mar. 2011, pp. 25–34.

[146] Eva Van Emden and Leon Moonen. "Java quality assurance by detecting code smells." In: *Working Conf. Reverse Eng.* 2001, pp. 97–106.

[147] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003, p. 235.

[148] Bartosz Walter and Blazej Pietrzak. "Multi-criteria Detection of Bad Smells in Code with UTA Method 2 Data Sources for Smell Detection." In: *Extreme Programming and Agile Processes in Softw. Eng.* 2005, pp. 154–161.

[149] WCER. *Transana [online] Available at: http://www.transana.org [Accessed 10 May 2012]*. 2012.

[150] Kurt Dean Welker. "Software Maintainability Index Revisited." In: *CrossTalk – Journal of Defense Software Engineering* (2001).

[151] Limei Yang, Hui Liu, and Zhendong Niu. "Identifying Fragments to be Extracted from Long Methods." In: *Asia-Pacific Software Engineering Conference*. 2009, pp. 43–49.

[152] Robert Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.

[153] Nico Zazworka, Victor R. Basili, and Forrest Shull. "Tool supported detection and judgment of nonconformance in process execution." In: *Int'l Symposium on Empirical Softw. Eng. and Measurement*. 2009, pp. 312–323.

[154] ZD Soft. *ZD Soft Screen Recorder [online] Available at: http://www.zdsoft.com [Accessed 10 May 2012]*. 2012.

[155] Min Zhang, Tracy Hall, and Nathan Baddoo. "Code Bad Smells: a review of current knowledge." In: *Journal of Software Maintenance and Evolution: Research and Practice* 23.3 (2011), pp. 179–202.

Part 2: List of Papers

Paper 3

Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data

Author: Aiko Yamashita

Abstract

Code smells are indicators of deeper design problems that may cause difficulties in the evolution of a software system. This paper investigates the capability of twelve code smells to reflect actual maintenance problems. Four medium-sized systems with equivalent functionality but dissimilar design were examined for code smells. Three change requests were implemented on the systems by six software developers, each of them working for up to four weeks. During that period, we recorded problems faced by developers and the associated Java files on a daily basis. We developed a binary logistic regression model, with "problematic file" as the dependent variable. Twelve code smells, file size, and churn constituted the independent variables. We found that violation of the *Interface Segregation Principle (a.k.a. ISP Violation)* displayed the strongest connection with maintenance problems. Analysis of the nature of the problems, as reported by the developers in daily interviews and think-aloud sessions, strengthened our view about the relevance of this code smell. We observed, for example, that severe instances of problems related to change propagation were associated with ISP Violation. Based on our results, we recommend that code with ISP Violation should be considered potentially problematic and be prioritized for refactoring.

Keywords – Software Maintenance, Code Smells, Refactoring, Maintenance Problems

1 Introduction

The presence of "smells" in the code may degrade quality attributes such as understandability and changeability and lead to a higher likelihood of introduction of faults. In short, code smells are symptoms of potentially problematic code from a software maintenance/evolution perspective. In [16], twenty-two code smells are described, each of them associated with refactoring strategies that can be applied to prevent potentially negative consequences of "smelly" code. However, code smells are only indicators of problematic code. Not all of them are equally harmful and some may not be harmful at all in some contexts. In addition, refactoring implies a certain cost and risk, e.g., any changes made in the code may induce unwanted side effects and introduce faults in the system. Consequently, we need to better understand the capability of code smells to explain maintenance problems and to identify the code smells that are likely to be the best indicators of such problems. This would enable better prioritization of the most influential refactorings to improve the maintainability of a system.

Previous studies have investigated the relationship between different code smells and different maintenance outcomes (e.g., effort, change size and defects, see Section 2.2). A reasonable assumption in these studies is that more effort, larger changes, and increased defects would imply lower maintainability. In this paper, a different approach is followed: we use the presence of *maintenance problems* experienced by developers, as our measure of maintainability. This approach may have its limitations (e.g., maintenance problems may differ in their severity), but it may also provide several advantages compared to the previously used outcome-based measures. In particular, we believe that observing maintenance problems is a more direct measure of the aim of code smells (i.e., to detect problematic code), than predicting effort, change size and defects. Fowler's [16] descriptions of code smells depict situations, where certain characteristics in the code make it difficult or *problematic* to understand, modify or test code; those problems may in turn have negative consequences in the effort, change size and defects. Consequently, there is a causal "step" in-between code smells and maintenance outcomes that needs to be investigated.

This paper reports on a study where we examined the presence of twelve code smells in four Java systems. The systems were the object of several change requests for a period of four weeks. During that period, we recorded problems faced by developers and the associated Java files on a daily basis. The maintenance problems were recorded in detail from interviews and think-aloud sessions with the developers. A binary logistic regression model was developed with the variable "problematic file" as the dependent variable. The presence of different types of code smells together with other essential properties of the file and the task constituted the independent variables. Further, the qualitative data from

interviews and think-aloud sessions were analyzed in order to support the results from the regression model and to shed a light on *how* certain code smells can cause problems during maintenance.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 describes the study design, including a description of the systems under analysis and the maintenance tasks. Section 4 presents the results. Section 5 presents the discussion of the results. Section 6 summarizes our findings and presents plans for future work.

2 Related Work

2.1 Code Smells

Code smells have become an established concept in patterns or aspects of software design that may cause problems in the further development and maintenance of software systems [16, 25, 42]. Code smells are closely related to Object-Oriented design principles, heuristics and design patterns, see for example principles and heuristic in [50] and [8], and design patterns in [17, 7, 26]. Van Emden and Moonen [55] provided, as far as we know, the first formalization of code smells and described a detection tool for Java programs. Most of the current detection approaches for code smells are automated. Examples of these approaches can be found in [36, 35, 40, 39, 41, 49, 2, 21, 42]. Work on automated detection of code smells has led to a range of detection tools such as Borland Together [6], InCode [18], JDeodorant [15, 54] and iSPARQL [22]. The analysis in this paper uses the tools Borland Together and InCode.

2.2 Empirical Studies on Code Smells

Zhang et al. [60] conducted a systematic literature review to describe the state of art in code smells and refactoring, based on conferences and journal papers from 2000 to June 2009. They found that very few publications conducted empirical studies on code smells, and that focus is mostly centered on developing new tools and methods for supporting automatic detection of code smells.

Mäntylä et al. [33, 32] conducted an empirical study of subjective detection of code smells. The study compared subjective detection with automated metrics-based detection and found that the results depend on developers' experience level. The authors reported that experienced developers identified more complex code smells and that increased experience with a module led to less reported code smells. Mäntylä et al. [31] also found a high degree of agreement between developers when reporting on the presence of simple code

smells, but quite low agreement when asked to recommend refactoring decisions. Previous studies have investigated the effects of individual code smells on different maintainability related aspects, such as *defects* [43, 28, 19, 9, 48], *effort* [11, 10, 29, 1] and *changes* [23, 21, 45].

D'Ambros et al. [9] analyzed code from seven open source systems and found that neither Feature Envy nor Shotgun Surgery code smells were consistently correlated with defects across systems. Juergens et al. [19] analyzed the proportion of inconsistently maintained Duplicated Code in relation to the total set of duplicated code in C#, Java, and Cobol systems. They found (with the exception of Cobol systems) that 18 percent of the inconsistently maintained duplicated code was associated with faults. Li and Shatnawi [28] investigated the relationship between six code smells and class error probability in an industrial system and found the presence of Shotgun Surgery to be connected with a statistically significant higher probability of faults. Monden et al. [43] performed an analysis of a Cobol legacy system and concluded that the cloned modules were more reliable, but demanded more effort to maintain than non-cloned modules. Rahman et at. [48] conducted a descriptive analysis and non-parametric hypothesis testing of source code and bug trackers in four systems. Their results suggest that clones tend to be less defect-prone than non-cloned code in general.

Abbes et al. [1] conducted an experiment in which twenty-four students and professionals were asked questions about the code in six open source systems. They concluded that God classes and God methods in isolation had no effect on effort or quality of the responses, but when appearing together they led to a statistically significant increase in response effort and a statistically significant decrease in percentage of correct answers. Deligiannis et al. [11] conducted an observational study where four participants evaluated two systems, one compliant and one non-compliant to the principle of avoiding God classes. Their main conclusion is that familiarity with the application domain plays an important role when judging the negative effects of a God class. They also conducted a controlled experiment [10] with twenty-two students as participants. Their results suggest that a design without a God class will result in more completeness, correctness and consistency compared to designs with a God class.

Lozano and Wermelinger [29] compared the maintenance effort of methods during periods when they did not contain a clone and when they did contain a clone. They found that there was no increase in the maintenance effort in 50 percent of the cases. However, when there was an increase in effort, this increase could be substantial. They reported that the effect of clones on maintenance effort depends more on the areas of the system where the clones are located than the cloning itself. Khomh et al. [21] analyzed the source code of Eclipse IDE, and they found that in general, Data classes were changed more often

than non-Data classes. Kim et al. [23] reported on the analysis of two medium-sized open source libraries (Carol and dnsjava). They reported that 36 percent of the total amount of duplicated code in the system needed to be simultaneously updated as a consequence of the product evolution. The rest of the duplicated code evolved independently and did not require simultaneous updates. Olbrich et al. [45] reported on an experiment involving the analysis of three open-source systems. They observed that God class and Brain class code smells were changed less frequently and had fewer defects than other classes when adjusting for differences in the class size.

From the identified empirical studies in code smells, it is possible to observe that not all code smells are equally harmful. Also, they are not consistently harmful across studies, indicating that their effects are potentially contingent on contextual variables or interaction effects. For example, the study by Li and Shatnawi [28] found that the presence of Shotgun Surgery leads to faults. D'Abros et al. [9] on the other hand, found no such connection between Shotgun Surgery and faults. Results from studies on duplicated code suggest that the effect of duplication depends of factors such as the programming language (e.g., results from Cobol system differed from the other types of systems in the study by Juergens et al. [19]. Similarly, results from studies on God class seem to give different results. Deligiannis et al. [11] reported that God class indicated problems, while Abbes et al. [1] concluded that a God class in isolation is not harmful. Olbrich et al. [45] reported that God class was less connected with problems in regression models when file size is used as a covariate.

2.3 Motivation of the Study Design

The design choice for this study is based on the assumption that observing the actual problems developers face during maintenance can provide a good picture of the role that code smells play in maintenance. Previous studies of code smells have mainly focused on duplicated code and a few other code smells. We aim to expand on the set of code smells studied, in order to better understand their respective explanatory capabilities. We believe that such insight can support refactoring prioritization endeavors. Our focus on problematic code may be closer to the original descriptions of code smells given by Fowler [16], since code smell definitions are more closely associated with *problematic maintenance* rather than indirect measures such as change effort, change size, and defects. To the best of our knowledge, analyzing the connection between maintenance problems and a large set of code smells has never been conducted. To extend our ability to detect causal relationships in our study, we decided to support the quantitative results with qualitative observations. For these reasons, the study includes data from interviews, recording of the developers' behavior and data from think-aloud sessions.

3 The Empirical Study

3.1 The Systems Maintained

In 2003, Simula Research Laboratory's Software Engineering department sent out a tender for the development of a web-based information system to keep track of their empirical studies. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all of them used the same requirements specification. More details on the original development projects can be found in [3]. The four development projects led to four systems with the same functionality. We will refer to the four systems as System A, System B, System C, and System D in this study. The systems were primarily developed in Java and they all have similar three-layered architectures. Although the systems are comprised of nearly identical functionality, there were substantial differences in how the systems were designed and coded. Table 3.1 shows the differences in lines of code (LOC) per system.

Table 3.1: Size of the systems analyzed

	System A	**System B**	**System C**	**System D**
LOC	7937	14549	7208	8293

The systems were all deployed in Simula Research Laboratory's Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to the database in the CMS in order to access data related to researchers, studies, and publications, i.e., to extract the information needed for the purpose of keeping track of the empirical studies.

3.2 The Maintenance Tasks and the Developers

In 2008, Simula Research Laboratory introduced a new CMS called *Plone* [47] and, consequently, it was no longer possible for the systems to remain operational. This situation required that the systems be adapted to this new environment. The adaptive task, together with additional functionality required by the users, constitute the maintenance tasks reported in this paper. Two Eastern European software companies were contracted to conduct the maintenance tasks. They completed the tasks between September and December 2008, at a cost of approx. 50,000 Euros. The maintenance tasks, which are briefly described in Table 3.2, were completed by six different developers. All developers completed all three maintenance tasks individually. The developers were recruited from a pool of 65 participants from a previously completed study on *programming skill* [5]. All

the selected developers were evaluated to have good development skills. This is defendable given that they all scored better than average skill. More about the skill scores used for this purpose can be found in [5]. All developers were deemed to have sufficient English skills for the purpose of our study.

Table 3.2: Maintenance tasks carried out during the study

No.	Task	Description
1	Adapting the system to the new Simula CMS	The systems in the past had to retrieve information through a direct connection to a relational database within Simula's domain (information on employees at Simula and publications). Now Simula uses a CMS based on the Plone platform, which uses an OO database. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications; a String type is used now. Task 1 consisted of modifying the data retrieval procedure by consuming a set of web services provided by the new Simula CMS in order to access data associated with employees and publications.
2	Authentication through webservices	Under the previous CMS, authentication was done through a connection to a remote database using authentication mechanisms available at that time for the Simula web site. Task 2 consisted of replacing the existing authentication by calling a web service provided for this purpose.
3	Add new reporting functionality	The devised functionality provided options for configuring personalized reports, where the user could choose the type of information related to a study to be included in the report, define inclusion criteria according to researchers who were in charge of the study, sort the resulting studies according to the date that they were finalized, and group the results according to the type of study. The configuration should be stored in the system's database and only be editable by the owner of the report configuration.

3.3 Study Design

The Process

First, the developers were given an overview of the tasks (e.g., the motivation for the maintenance tasks and the expected activities). Then they were given the specifications of the three maintenance tasks. When needed, they would discuss the maintenance tasks with the researcher (the author of this paper) who was present at the site during the entire project duration. Daily meetings with the developers were conducted to track the progress and the problems encountered. Thirty minute think-aloud sessions were conducted every second day, and performed at random points of the development work. Acceptance tests and individual open interviews, with a duration of 20-30 minutes, were conducted once all tasks were completed. In the open-ended interviews, the developers were asked about their opinions of the system (e.g., about their experiences when maintaining it). Eclipse was used as the development tool, together with MySQL [46] and Apache Tomcat [52]. Defects were registered in Trac [13]; Subversion or SVN [51] was used as the versioning system. A plug-in for Eclipse called Mimec [27] was installed in each developer's computer in order to log all the user actions performed at the GUI level with milliseconds precision.

After completing the three maintenance tasks on one system, they repeated the same maintenance tasks on a second system. The developers varied with respect to which of the four systems we assigned as the first and the second system (designated as "first round" and "second round" systems, respectively). Clearly, there is a learning effect from repeating the same tasks on a second system. This learning effect does, however, also reflect quite a realistic situation where the developers have relevant experience, i.e., have completed quite similar tasks before. In addition, our study was designed to be quite robust with respect to this learning effect by focusing on whether a file was identified as problematic by at least one developer in *at least one of the rounds*. More details on this will be given in the next section.

The Dependent Variable

One aim of the study was to build an explanatory model of maintenance problems. This model consisted of dependent variable, i.e., the variable we try to explain, and independent variables, i.e., the variables used to explain the dependent variable. The *dependent variable* of our model was related to whether a Java file was perceived as problematic or not by at least one of the developers in at least one of the rounds. In the context of this study, maintenance-related problems were interpreted as "any struggle, hindrance, or problem developers encountered while they performed their maintenance tasks, which were possible to observe through daily interviews and think-aloud sessions."

The daily interviews with each developer enabled the recording of problems encountered during maintenance while they were still fresh in their mind. The following is an example of a comment given by one developer, who complained about the complexity of a piece of code: "It took me 3 hours to understand this method..." Such types of comments were used as evidence that there were maintenance (understandability) problems in the file that included this method.

During the think-aloud sessions, the developers' screens were recorded with a Screen Recorder program [59]. Sometimes the maintenance problems were derived from more than one data source (e.g., combination of direct observation, the developers' statements on a given topic/element, and the time/effort spent on an activity). Since not all maintenance problems were associated with a piece of source code, some maintenance problems were not included in the model building. When it was possible to map the identified maintenance problems to a file, that file was categorized as problematic.

An example of the process to collect and structure data related to the variable "problematic file" is given in Table 3.3. In this example, the observations by the researcher and the statements from the developer lead to the conclusion that the initial strategy of replacing several interfaces in order to complete Task 1 was not feasible due to unmanage-

able error propagation. The developer spent up to 20 minutes trying to follow the initial strategy (i.e., replace the interfaces), but decided to rollback and to follow an alternative strategy (i.e., forced casting in several locations) instead. As a result of this information (i.e., problems due to change propagation), the files containing the interfaces were deemed problematic.

While the assessment of problematic files can be subjective to some extent, the connections between problems and code in this study were deemed to be quite direct. In some cases it would have been meaningful to classify the severity of maintenance problems. However, in some other cases this would have lead to quite subjective assessments. Consequently, in order to avoid a more complex model and increased subjectivity in the interpretations, it was decided to treat "problematic file" as a binary variable.

A logbook was kept during the interviews and think-aloud sessions, in which the maintenance problems were registered in detail. For each identified maintenance problem, the following information was collected:

a. The developer and the system.

b. The statements given by the developers related to the maintenance problem.

c. The source of the problem (e.g., whether it was related to the Java files, the infrastructure, the database, external services)

d. List of files/classes/methods mentioned by the developer when talking about the maintenance problem.[1]

In short, the categorization of the problematic files was based on either direct observations of the developers' behavior during the think-aloud sessions or based on comments made by the developers during the daily interviews. In addition to the categorization of a file as problematic or not, qualitative analysis was conducted on the observed or reported maintenance problems. This analysis was based on *explanation building technique* [58] and aimed at determining the extent to which the maintenance problems were caused by, or only correlated with, code smells. An essential input to the qualitative analysis was the analysis of Java files that were modified or inspected during the maintenance work. These files were identified by using the logs generated by Mimec [27]. This plug-in recorded not only the type of action performed by the developer in the IDE, but also the Java element (if any) that was the subject of the interaction, such as the name of the file selected, or the name of the class/method being edited. For more details on how the Mimec logs we processed and analyzed, see [57].

[1]Note: Consequently, that one maintenance problem could be related to several problematic Java files

[2]A persistence framework is used as part of Java technology for managing relational data (more specifically data entities). For more information on Java persistence, see www.oracle.com

Table 3.3: Excerpt from a think-aloud session

Code	**Statement/Action by Developer**	**Observation / Interpretation**
Goal	Change entities' ID type from Integer to String	This is part of the requirements for Task 1.
Finding	"Persistence is not used consistently across the system, only few of them are actually implementing this interface so..."	Persistence[2] is referred to as two interfaces for defining business entities, which are associated with a third-party persistence library, and is not used consistently in the system.
Strategy	"I will remove this dependency, I will remove two methods from the interface (getId an setId) added for integer and string. This strategy forces me to check the type of the class, but this is better than having multiple type forced castings throughout the code."	Developer decides to replace two methods of the Persistence interface (i.e., getId() an setId()), which are using Integer, with methods with String parameters.
Action	Engages in the process of changing id in interface PersonStatement.java	Developer engages in the initial strategy.
Muttering	"Uh, updates? just look at all these compilation errors..."	Developer encounters compilation errors after replacing the methods in the interfaces.
Action	Fix, refactor, correct errors.	Starts correcting the errors.
Strategy	"Ok... I need to implement two types of interfaces, one for each type of ID for the domain entities. I will make PersistentObjectInt.java for entities that use Integer IDs and PersistentObjectString.java for String IDs."	Change of strategy, decides to actually replace the interface instead of replacing the methods in the interface.
Action	Fix more errors from Persistable.java.	More compilation errors appear.
Action	Continue changing interface of the entity classes into PersistentObjectInt and PersistentObjectString.	Attempt to continue with the second strategy.
Action	(After 20 minutes) Rollback the change.	Developer realizes that the amount of error propagation is ummanegeable so rolls back the changes.
Muttering	"Hmm... how to do this?"	Developer thinks of alternative options.
Strategy	"Ok, I will just have to do forced casting for the cases when the entity has String ID."	Developer decides to use the least desirable alternative: forced type castings whenever they are required.

The Independent Variables

Twelve code smells were extracted from the systems by using Borland Together [6] and InCode [18] and used as independent variables in the regression model. Table 3.4 presents descriptions of the code smells detected in the systems (taken from [16, 37]), and their respective scale types. The *detection strategies* used in the tools were mostly based on the work by Marinescu [34] (See Appendix A), who proposed combinations of different code metrics to detect code smells.

A design principle violation called *Interface Segregation Principle Violation* (ISP violation) was also included (See [37]). This design violation was included because it was deemed as a potentially important indicator of maintenance problems and because Borland Together was able to detect it. ISP Violation is not part of the twenty-two code smells defined by Fowler and Beck, but it can be considered a code smell since it constitutes an anti-pattern believed to have negative effects on maintainability [37].

As can be seen from Table 3.4, all code smells except for Feature Envy were treated as binary variables. This was done because most of the code smells are binary by nature (i.e., present = 1, not present = 0), and also because the majority of the non-binary code smells displayed only 1 to 2 instances per file in the study. This means that it would have not been possible to gain much in explanatory power by increasing the complexity of the model to include the amount of observations of a code smell in a file. Natural logarithm was applied to the Feature Envy variable to avoid a too strong effect from a few very high values.

In addition to the code smell variables, a variable reflecting the file size (LOC including comments and blank lines), and a variable reflecting the size of the task (churn) on a file were included. Churn was measured as the sum of lines of code inserted, updated, and deleted in a file. These variables were measured using SVNKit [53], a Java library for requesting information to Subversion. The variables file size and churn were included in the regression model to adjust for an increase of likelihood of a file being perceived as problematic, not because of the presence of code smells, but because of a large size or a large update of the file. Both variables were log-transformed to avoid large influences from a few very high values.

There could be differences in the effect of code smells on problematic code depending on which of the four systems were under development. Therefore, *system* was also included as a (nominal) control variable in the model. The way we defined "problematic file" (i.e., whether a Java file has been perceived as problematic or not by at least one of the developers in at least one of the rounds) means that we did not need to include "developer" or "round" as independent variables in the model.

Table 3.4: Code smells and their descriptions from [16, 37]

Code Smell (ID)	**Description**	**Variable Type**
Data Class (DC)	Classes with fields and getters and setters not implementing any function in prticular.	Binary
Data Clumps (CL)	Clumps of data items that are always found together whether within classes or between classes.	Binary
Duplicated Code in conditional branches (DUP)[3]	Same or similar code structure repeated within the branches of a conditional statement.	Binary
Feature Envy (FE)	A method that seems more interested in others classes than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.	Continuous
God Class (GC)	A class has the God Class code smell if the class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles.	Binary
God Method (GM)	A class has the God Method code smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method.	Binary
Misplaced Class (MC)	In "God Packages" it often happens that a class needs the classes from other packages more than those from its own package.	Binary
Refused Bequest (RB)	Subclasses do not want or need everything they inherit.	Binary
Shotgun Surgery (SS)	A change in a class results in the need to make a lot of little changes in several classes.	Binary
Temporary variable is used for several purposes (TMP)	Consists of temporary variables that are used in different contexts, implying that they are not consistently used. They can lead to confusion and introduction of faults.	Binary
Use interface instead of implementation (IMP)	Castings to implementation classes should be avoided and an interface should be defined and implemented instead.	Binary
Interface Segregation Principle Violation (ISPV)	The dependency of one class to another should consist on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling.	Binary

The Analysis

First, an exploration of the maintenance problems was conducted. Then, a binary logistic regression model was built using the independent variables to explain the dependent variable (i.e., likelihood of a file being problematic). The results of the regression model were complemented by exploratory factor analysis, that is Principal Component Analysis (PCA), in order to investigate the clusters of code smells, (i.e., to what degree different code smells were correlated with others). This may be helpful to understand the nature and potential effects of clusters of code smells. Finally, a qualitative follow-up analysis was conducted based on the results from the regression analysis. The data set used in the regression analysis was based on recording a data point each time a developer read or updated a file. The way the dependent variable was defined (i.e., that a file was categorized as problematic when at least one developer had had maintenance problems that could be related to that file), would result in a substantial amount of data having the same values for all variables except the size of the task on that file (i.e., churn) when using the original data set. To avoid this strong degree of dependency among the observations, all "duplicates" (i.e., all highly dependent observations) were removed, and the average churn of all the recordings belonging to the same file was used to represent the typical size of the update of that file. This led the data set to have a maximum of one data point per Java file, which either had the value 1 (problematic) or 0 (not problematic). Consequently, this approach increased the robustness of the model and reduced the problem of dependency between the observations.

4 Results

4.1 Exploration of the Maintenance Problems

Most of the maintenance problems identified were related to: (1) introduction of defects as result of changes (25%), (2) time-consuming changes (39%), and (3) troublesome program comprehension and information searching (27%). Table 3.5 provides a description of each type of problem.

In total, 137 different maintenance problems[4] were identified. From the total number of maintenance problems, 64 (47%) related to Java source code. The remaining 73 (53%) constituted problems not directly related to code (e.g., lack of adequate technical infras-

[3]Note that this code smell is not the actual duplicated code, but a local version of it, only located across conditional branches. This code smell was included because Borland Together could detect it. Analysis of other types of duplicated code is beyond the scope of this study.

[4]A more complete description of the nature and distribution of the different problems identified during maintenance is available by sending a request to the author.

Table 3.5: Description of the three main types of problem identified

Type of problem	Description
Introduction of defects	Undesired behavior, or unavailability of functionality in the system (i.e., defects) manifested after modifying different components of the system. This introduced delays in the project, and forced developers to rollback initial strategies for solving the tasks on several occasions.
Time-consuming or costly changes	Time consuming or costly changes were associated with two main situations: 1) the high number of components in the system that needed changes for accomplishing a task, made the overall task time-consuming, and 2) the presence of cognitively demanding problems to be solved, or intrincate design, visualization or distribution of information made the task difficult and, consequently, time-consuming.
Troublesome program comprehension and information searching	This problem type is comprised of three situations: 1) struggle while trying to get an overview of the system or while trying to achieve high-level understanding of the system's behaviour, 2) during low-level understanding of the code, developers become confused because they find inconsistent or contradictory evidence in different components of the system, and 3) developers struggle while searching for information or the "task context" (e.g., finding the place to perform the changes and/or finding the data needed to perform a task).

tructure, developer coding habits, external services, runtime environment, and defects initially present in the system).

The high percentage of non-source code related problems suggests that problems identifiable via current definitions of code smells may only cover a smaller part (in this case 47%) of the total problems identified during maintenance. This indicates that there are substantial limitations in the use of source code analysis to explain maintenance problems. This may also imply that alternative evaluation methods should be used in combination with code smell analysis, in order to achieve a comprehensive maintainability evaluation.

In total, 301 Java files across all four systems were modified or inspected by at least one developer during maintenance. Out of those files, 61 (approx. 20%) of them were reported as problematic during maintenance by at least one developer. Table 3.6 presents the numbers and proportions of problematic files across the four systems.

Table 3.6: Distribution and percentage of problematic vs. non-problematic files

System	Problematic = 1		Problematic = 0		N
A	11	20%	45	80%	56
B	37	30%	88	70%	125
C	3	12%	22	88%	25
D	10	11%	85	89%	95
Total	61	20%	240	80%	301

In Table 3.1, we reported that System B was about twice as large as the other systems, and that the other systems had about the same number of lines of code. Table 3.6 shows that System B was also the system with the largest proportion of problematic files. Considering that the four systems implemented the same functionality and were subject to the same

Table 3.7: Percentage of the files inspected or modified during maintenance that contained any of the code smells investigated

Sys	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	TMP	IMP
A	0.18	0.11	0.02	Mean: 0.36 SD: 0.501	0.02	0.07	0.11	0	0.27	0.12	0.16	0.05
B	0.2	0.01	0.02	Mean: 0.12 SD: 0.37	0.04	0.08	0.06	0.02	0.06	0.12	0.1	0.02
C	0.32	0.12	0.04	Mean: 0.14 SD: 0.578	0.12	0.08	0.04	0	0	0	0.16	0
D	0.22	0.05	0.02	Mean: 0.1 SD: 0.36	0.02	0.04	0.12	0.02	0.01	0.13	0.03	0
All	0.21	0.05	0.02	Mean: 0.16 SD: 0.423	0.04	0.07	0.09	0.01	0.08	0.11	0.09	0.02

maintenance tasks, this may suggest that writing more lines of code to implement the same piece of functionality can indicate an increase in maintenance problems.

Table 3.7 presents the percentage of files inspected or modified during maintenance that contained any of the investigated code smells. In the case of Feature Envy, since it is treated as a continuous variable, the mean and standard deviation are presented. As can be seen in Table 3.7, some code smells seldom occur in the systems (e.g., use of implementation instead of interface (IMP), Misplaced Class (MC)). Consequently, for those code smells, it is unlikely that the regression model would report coefficients significantly different from zero, unless the effect size is very large.

4.2 The Binary Logistic Regression Model

A binary logistic regression model was built with all the variables entered in a single step. The overall fit of the model is indicated by the values displayed in Table 3.8. The chi-square statistics of the -2 Log likelihood suggests that our model performs significantly better than the null model, i.e., a model always predicting the most common outcome. The R^2 values (Cox & Snell, and Nagelkerke) provides further support that our model has a good fit, but do also shows that, not unexpectedly, a substantial part of the variance is not explained by our model. The Hosmer and Lemeshow test gives a R^2 of 0.864, and further supports that the model provides a good fit of the data.

Table 3.8: Model test

-2 Log likelihood	**DF**	**ChiSquare**	$Prob > ChiSq$
223.689	17	79.758	.000
			Cox & Snell R^2: .233
			Nagelkerke R^2: .367
			Hosmer & Lemeshow R^2: .864

Table 3.9 shows the percentage of correctly and incorrectly classified cases from the model. The performance measures for the model are: *accuracy* = 0.847, *precision* = 0.742, and *recall* = 0.377. As can be seen, the classification performance is far from perfect, but it is much better than the random model or the null model that always classifies a file as not problematic. This provides further evidence for the meaningfulness of the proposed explanatory regression model.

Table 3.9: Classification performance

Observed		**Classified**		
		'problematic?'		Percentage Correct
		0	1	
'problematic?'	0	232	8	96.7
	1	38	23	37.7

Table 3.10 displays the independent variables of the model with their coefficients (B), standard errors (S.E.), Wald statistic (Wald), significance levels (Sig.), odds ratios (Exp(B)), and lower and upper confidence intervals for the odds ratios. All the variables for which the odds ratios are statistically significant, here defined as $p < .05$, are in bold and their significance levels are marked with an asterisk. The Wald statistic indicates if the coefficient is significantly different from zero. The odds ratio indicates the change in odds resulting from a unit change in the explanatory variable given that all other variables are held constant.

As can be seen in Table 3.10, a file belonging to System B, which has an odds ratio of 4.137, has a significantly higher likelihood of being connected with maintenance problems than a file belonging to the other systems. This suggests, similar to what we observed earlier, that the size of the system might be quite important when explaining maintenance problems.

Regarding the code smells, we can see that the odds ratio for the code smell ISP Violation is the largest (Exp(B) = 7.610, p = .032), which suggests that ISP Violation is able to explain much of the maintenance problems. This model also find the Data Clump code smell as a significant contributor to the model (Exp(B) = 0.053, p = .029), but contrary to ISP Violation, this code smell indicates fewer maintenance problems.

Table 3.10: Model variables

Variable	B	S.E.	Wald	Sig.	Exp(B)	95% C.I.for Exp(B)	
						Lower	Upper
System A	.188	.612	.094	.759	1.207	.364	4.005
System B	**1.420**	**.481**	**8.717**	**.003***	**4.137**	**1.612**	**10.617**
System C	-.361	.895	.163	.687	.697	.121	4.027
DC	.036	.479	.006	.940	1.037	.405	2.650
CL	**-2.935**	**1.340**	**4.796**	**.029***	**.053**	**.004**	**.735**
DUP	-2.721	1.747	2.427	.119	.066	.002	2.018
FE	-.001	.493	.000	.999	.999	.380	2.627
GC	.605	1.180	.263	.608	1.831	.181	18.494
GM	-.810	.916	.782	.377	.445	.074	2.679
ISPV	**2.029**	**.948**	**4.587**	**.032***	**7.610**	**1.188**	**48.749**
MC	1.151	1.325	.755	.385	3.162	.236	42.418
RB	.231	.633	.133	.716	1.260	.364	4.359
SS	-.654	.778	.705	.401	.520	.113	2.392
TMP	.089	.659	.018	.892	1.093	.301	3.977
IMP	.311	1.108	.079	.779	1.365	.156	11.978
Churn	**.723**	**.152**	**22.694**	**.000***	**2.061**	**1.531**	**2.775**
LOC	.189	.272	.482	.487	1.208	.709	2.058
Constant	-4.237	1.162	13.305	.000	.014		

This study focused on maintenance problems and did not collect much information about what could lead to fewer maintenance problems. The strong, positive effect of the Data Clump suggests, however, that it is a candidate for further studies, i.e., the results from this study suggests that there could be types of Data Clump that increase rather than decrease the maintainability of software systems. The coefficient of the variable churn is significantly different from zero ($p < .001$), and the odds ratio is higher than one. This is not surprising, given that the more work completed on a file, the more likely is it that there will be some problem connected with that file.

Multicollinearity may lower the robustness of the interpretation of the individual coefficients of our model, which does include variables likely to be correlated. However, when reducing the multicollinearity by representing each factor (see the principal component analysis in Section 4.3) by one of its code smells, the coefficients did not change very much. Also, multicollinearity diagnostics was conducted, where the collinearity statistics Tolerance and VIF were calculated for the variables in the model. [38] suggests that a Tolerance value less than 0.1 almost certainly indicates a serious collinearity problem, and [44] suggests that a VIF value greater than 10 is cause for concern. All the predictors displayed Tolerance values over 0.4 and the VIF values ranged from 1.008 to 3.891. These results suggest that the presence of multicollinearity is not a major problem in this regression analysis.

4.3 The Factor Analysis

A principal component analysis (PCA) was conducted on the 301 data points using orthogonal rotation (varimax). The Kaiser-Meyer-Olkin measure verified the sampling adequacy for the analysis, KMO = .604, and all KMO values for individual items were > .5, which is above the acceptable limit according to [20]. Bartlett's test of Sphericity $\chi^2(66) = 561.252$, $p < .001$, indicated that the correlations between the items were sufficiently large for PCA. An initial analysis was run to obtain eigenvalues for each component in the data. Five components had eigenvalues over Kaiser's criterion of 1 and in combination explained 63.5% of the variance (See Table 3.11).

Table 3.11: Total variance explained

Component	**Init. Eigenvalues**			**Extraction Sums of Squared Loadings**			**Rotation Sums of Squared Loadings**		
	Total	Var.%	Cum.%	Total	Var.%	Cum.%	Total	Var.%	Cum.%
1	2.442	20.350	20.350	2.442	20.350	20.350	2.315	19.291	19.291
2	1.768	14.731	35.081	1.768	14.731	35.081	1.693	14.108	33.399
3	1.305	10.875	45.956	1.305	10.875	45.956	1.462	12.180	45.579
4	1.073	8.942	54.898	1.073	8.942	54.898	1.098	9.147	54.725
5	1.033	8.607	63.505	1.033	8.607	63.505	1.054	8.779	63.505
6	.885	7.378	70.883						
7	.826	6.881	77.764						
8	.767	6.389	84.153						
9	.650	5.415	89.568						
10	.554	4.613	94.181						
11	.406	3.385	97.566						
12	.292	2.434	100.000						

Table 3.12 shows the factor loadings after rotation. When observing the factors, we see that the code smells God Method and God Class are the closest in Factor 1, followed by the code smells Temporal variable used for several purposes, duplicated code in conditional branches, and Feature Envy. Given that the detection strategies of the first two code smells are based on size measures, it is natural that they appear together. Also, large classes often use many different variables, which increase the chances of the presence of Temporary variable used for several purposes. Feature Envy is also present when there are complex methods that need many parameters from other classes. Factor 1 variables may, consequently, be considered to relate to the size of the code. ISP Violation and Shotgun Surgery belong together in a separate factor (Factor 2). This indicates that they may represent, to some extent, the same construct (e.g., related to wide-spread, afferent coupling). Also, they do not seem to relate much to the size of the code (Factor 1). In Section 4.4, we discuss the findings related to ISP Violation in the light of the

qualitative evidence gathered during the maintenance work. Data Class and Data Clump are together in one factor (Factor 3). Implementation instead of interface seems to appear very seldom in our dataset and does not relate to any of the other code smells (Factor 4). Refused Bequest and Misplaced Class constitute the last factor (Factor 5), where Misplaced Class has a negative loading. This indicates that Misplaced Class tends to be negatively associated with this factor[5].

Table 3.12: Factor loadings after rotation

	Component				
	1	2	3	4	5
GM	.751				
GC	.730				
TMP	.687				
DUP	.595				
FE	.537				
SS		.896			
ISPV		.823			
DC			.751		
CL			.721		
IMP				.823	
RB					.822
MC					-.548
Eigenvalues	2.442	1.768	1.305	1.073	1.033
% of variance	20.350	14.731	10.875	8.942	8.607

4.4 Problems Related to ISP Violation

In the model reported in Section 4.2, the coefficient of the ISP Violation variable is significantly different from zero, and this code smell has by far, the highest odd ratio (i.e., it shows the strongest connection to files identified as problematic). This section provides further analysis on ISP Violation. First, details are presented on the files that contained this code smell and that were deemed problematic (Section 4.4). Based on the qualitative data collected during the interviews and think-aloud sessions, a report on how this code smell caused different types of difficulties for developers during maintenance is presented (Section 4.4). Finally, two significant cases observed during the study are described, where the interaction of ISP Violation with other code characteristics had acute consequences for maintainability (Section 4.4). The process of selecting the observations for Sections 4.4 and 4.4 were based on an examination of the qualitative information, resulting in several

[5]Positive and negative loadings can be associated with the same factor. For example, in surveys, negative loadings are caused by questions that are negatively oriented to a factor. A combination of positive and negative questions is normally used to minimize an automatic response bias by the respondents [12].

in-depth descriptions of maintenance problems deemed to be related to ISP Violation. Out of these observations, we selected those deemed to be most representative and illustrative for the nature of ISP Violation-related problems among the software developers.

Problematic Files with ISP Violation

In total, 45 occurrences of maintenance problems related to source code were associated with files containing at least one code smell. As much as 23 (more than 50%) of these problems were associated with files displaying ISP Violation. Notice that there is a many-to-many relationship between maintenance problem and files. More specifically, one maintenance problem can be associated with several or no files, and one file can be associated with different observed occurrences of maintenance problems. In our case, many of the maintenance problems were related to the same file. This means that the number of unique problematic files with ISP Violation was much lower (12) than the number of maintenance problems related to ISP Violation (23). Table 3.13 displays the code smells for the files that displayed the ISP Violation. As can be seen, a high proportion of problematic files with solely the ISP Violation and the Shotgun Surgery (7 out of 12), further support that these two code smells as essential to explain a major part of the maintenance problems.

Table 3.13: Code smells for the problematic Java files with ISP Violation

File	Sys	DC	CL	DUP	FE	GC	GM	ISPV	MC	RB	SS	TMP	IMP	Total
StudyDatabase.java	A	0	0	0	7	0	1	1	0	0	1	1	1	12
StudySortBean.java	A	1	1	0	0	0	0	1	0	0	0	0	0	3
ObjectStatementImpl.java	B	0	0	0	0	0	0	1	0	0	1	0	0	2
Person.java	B	0	0	0	0	0	0	1	0	0	1	0	0	2
Simula.java	B	0	0	0	0	0	0	1	0	0	1	0	0	2
Table.java	B	0	0	0	0	0	0	1	0	0	1	0	0	2
DB.java	C	0	0	2	16	1	2	1	0	0	0	1	0	23
Nuller.java	D	0	0	0	0	0	0	1	0	0	1	0	0	2
StudyDAO.java	D	0	0	1	10	1	2	1	0	0	1	1	0	17
StudySDTO.java	D	1	0	0	0	0	0	1	0	0	0	0	0	2
WebConstants.java	D	0	0	0	0	0	0	1	0	0	1	0	0	2
WebKeys.java	D	0	0	0	0	0	0	1	0	0	1	0	0	2

Maintenance Problems Related to ISP Violation

The analysis on the qualitative data exposed a relationship between ISP Violation and some of the negative consequences entailed by *wide afferent coupling*. Note that the

detection strategy for ISP Violation (See 7) uses several afferent coupling measures such as Class Interface Width (CIW), and Clients Of Class (COC) [34].

An analysis of the data from the think-aloud sessions showed that when the developers *introduced faults* in files with wide afferent coupling (thus, displaying ISP Violation), the consequences of these faults manifested themselves across different components that depended on them. This situation caused much of the systems' functionality to stop working after changes, and in some cases, lead to unmanageable error propagation. As an illustration, the classes located in the files *Nuller.java*, *StudyDAO.java*, and *DB.java* (See Table 3.13) were found to propagate erratic behavior across different parts of the system and were associated with three maintenance problems reported by two developers.

Also, when changes were introduced to the abovementioned classes, we observed that adaptations or amendments were needed in other classes depending on the ISP Violators. This resulted in time-consuming change propagation. This situation also caused the introduction of defects (as developers sometimes would miss parts of the code that needed amendments), resulting in a time-consuming, and an error-prone process. These observations mainly came from the daily interviews when developers mentioned certain files one day (e.g., "I am working with files X, Y"), and the next day they reported that the changes in those files demanded changes in more areas of the code and would introduce delays in the project.

To further illustrate problems related to ISP Violation, we will present two observations from think-aloud sessions and daily interviews suggesting that problems with the developers' *program comprehension* are also related to this code smell.

The first observation relates to the presence of crosscutting concerns. In System A, the domain entity "Person" was being used within the "User" context and also within the "Employee" context. As such, the management of privileges/access, and information/-functionality related to employees constituted crosscutting concerns. The crosscutting concerns manifested in System A, with the entity *Person* (located in the file *People-Database.java*) being accessed by many segments of the system. According to developers, this made the identification of the relevant task context very difficult.

The second observation relates to the presence of inconsistent design (manifested in the class *StudySortBean*, in System A). A major reason why the developers found System A difficult to understand seems to be due to inconsistent and incoherent data and functionality allocation, which was considered 'not logical' by the developers (two developers literally stated that the design "did not make sense"). The class *StudySortBean* was initially employed as a *Bean*[6] to sort a given list of empirical studies and present them

[6]In J2EE environments, it is common to use *Bean* files as data transfer objects. Their counterparts, the *Action* files (which in turn contain the business logic) access the *Bean* files.

in a report to the user. Probably throughout the initial development phase (i.e., not the maintenance phase), *StudySortBean* class started to acquire more responsibilities that did not correspond to the class, and turned into an *Action* file. This would be a good example of what Martin [37] calls, "wider spectrum of dissimilar clients..". This file was originally a *Bean* file that should only contain data, but it ended up containing functionality. As a result, this file initially containing Data Class, acquired the ISP Violation.

Both the data and the functionality were called from many different classes, many of them unrelated. Since the allocation of the data and functionality seemed rather arbitrary to the developers, they got confused about the rationale of such design. This case is very interesting because ISP Violation is not the real cause of the problem (the real problem was the inadequate allocation of data and functionality); nonetheless, the definition of ISP Violation and subsequent detection strategy could identify this situation.

The complete set of maintenance problems related to ISP Violation (See Appendix B) provides further details that support the analysis presented in this section.

Interactions between ISP Violation and other Code Characteristics

The following two observations illustrate the maintenance problems caused by *interaction effects* involving ISP Violation, and we believe they are representative of the types of problems identified during the study.

The first case was observed in System B, and it was related to time-consuming changes and defects after initial changes. This type of problem was reported by all developers who updated System B. The developers who worked on System B wanted to replace two interfaces (located in the files *Persistable.java* and *PersistentObject.java*[7]) with one new interface to support a String ID type in order to complete Task 1. Recall that Task 1 consisted of modiying functionality that accesses external data. The external data employs String type identifiers, as opposed to Integer types used in the system. Replacing the interfaces was not possible since the entire logic flow was based on primitive types instead of of domain entities. Both interfaces were restrictive and were made under the assumption that the identifiers for objects would always be Integers, and thus defined accessor methods getId() and setId() with Integer types. Notice that these interfaces did not display any code smells.

The maintenance problems seemed to occur because several critical classes in the system implemented these two interfaces. Many of the classes that implemented these interfaces (e.g., *ObjectStatementImpl* in Table 3.13) displayed ISP Violation, which resulted in extensive ripple effects when modifying the interfaces. It was observed that after the

[7]These interfaces are part of the Persistence Framework. As explained previously, Persistence Framework is used as part of Java technology for managing relational data (more specifically data entities). For more information on Java persistence, see www.oracle.com.

developers modified the interfaces, this led to an extremely high number of compilation errors. This induced the developers to rollback the initial changes in those files (i.e., keep the interfaces untouched) and instead perform forced casting wherever a String type identifier was required. Most developers used a considerable amount of time trying to replace the interface, and they were forced to rollback and perform the forced casting. This is an example of how the presence of a code smell may intensify or spread the effects of certain design choices throughout the system. Since classes with wide afferent coupling dispersion (and thus, containing ISP Violation) were coupled with these interfaces, any changes to the interfaces would have the same impact as if they were performed in the classes with the wide afferent coupling.

The second case relates to the observation that all systems except for System B contained one single "Brain Class" that "hoarded" most of the logic and functionality in those systems (They were located in the files *StudyDatabase.java*, *DB.java*, and *StudyDAO.java*). These classes were very large in comparison to other classes in the system, displayed a wide spread of both afferent and efferent coupling, and demanded high amounts of changes. All three 'hoarders' displayed ISP Violation, because they displayed many incoming dependencies from different segments of the system. Because of their high level of efferent coupling, they also contained Feature Envy. They also contained God Method, which is commonly present in big, complex classes. The developers found it difficult to foresee the consequences of changes performed in the "hoarders", given the combination of their internal complexity and the high number of dependent classes. Changes in the "hoarders" were essential to the maintenance tasks, and they were time consuming since the developers first had to understand the logic they contained. Even after the changes were made, errors would manifest in different areas of the system, causing further delays to the project.

In general, the qualitative data seems to support the results from the regression model and suggests that ISP Violation is a common indicator of a wide range of maintenance problems, and this may be useful in identifying problematic areas of the code.

4.5 Analysis of Files Containing Data Clump

The only other statistically significant coefficient in the regression model was related to the code smell Data Clump. Surprisingly, the odds ratio of the variable related to this code smell suggests that its presence is connected with a low likelihood of observing a problematic file. This is supported by the fact that of the total set of 14 files modified by developers during maintenance that contained Data Clump, only one file was deemed as problematic. The file *StudySortBean.java*, located in System A (previously described in Section 4.4) contained Data Clump but also violated the Interface Segregation Principle.

As we explained earlier, it seems as if ISP Violation was the main reason why the file was considered problematic. Most of the files containing Data Clump also constituted Data Classes, as described in the PCA analysis (Section 4.3) and are used as data containers in the systems.

After a systematic search through all the qualitative data related to the Data Clump and, the possibly connected, Data Classes, it was not possible to find evidence explaining why these code smells were connected with a lower rather than a higher likelihood of maintenance problems. The lack of qualitative evidence points a limitation of the study design, i.e., the strong focus on "maintenance problems" when collecting the qualitative data. With this focus, it is possible that some evidence was missed on the potentially positive effects of some code smells. In addition, it was observed during the study that developers rarely talked about attributes at class level of which they were fond. Developers were found to be more prone to complain about classes rather than express positive comments. The results of this study suggest that some code smells could actually constitute beneficial attributes in the code, and as such, they represent an interesting topic for further studies.

5 Discussion of Results

5.1 Comparison with Related Work

Within the curent literature on code smells, no study has yet reported the potential negative effects of ISP Violation in maintenance projects. Nevertheless, the results of this study can be related to results by Li and Shatnawi [28], who reported that Shotgun Surgery was positively associated with faults. In our study, it was observed that many instances of ISP Violation were accompained by the Shotgun Surgery code smell, which suggests that the detection strategies of these code smells may uncover related types of design shortcomings.

It was also observed that in some cases, interaction between ISP Violation and other code smells (or other design properties), seemed to cause maintenance problems. We know of only one study (by Abbes et al. [1]) that reported on the interaction between code smells (i.e., between God Class and God Method) and none between ISP Violation and other code smells. The effects of code smell combinations seems to be a topic that deserves more attention. The importance of studying the interactions between code smells is further supported by observations that, in some large classes, the maintenance problems may not be directly caused by the actual size of the class, but rather are a result of interaction effects across different code smells that happen to appear together in the same file (See our discussion in Section 4.4). Olbrich et al. [45] reported that when normalized

with respect to size, classes constituting God Class or Brain Class had less faults and demanded less effort. This means that not all God Classes or Brain Classes are harmful, but it would require additional indicators to determine how harmful they could be. An option, which we think deserves more attention, is to use the presence of ISP Violation to discriminate between instances of God Class that are and are not likely to be harmful.

In the model proposed in this paper, Data Clump is negatively associated with maintenance difficulties. This suggests that there are code smells that, at least in some contexts, may lead to a lower risk of maintenance problems. Li and Shatnawi [28] reported that Data Classes were not significantly associated with faults. This may be the same as what was found here, given that the code smells Data Class and Data Clump have a tendency to appear in the same file as shown in the factor analysis.

5.2 Threats to Validity

The validity of the study is considered and presented from three perspectives:

Construct Validity. The code smells were identified via detection tools to avoid subjective bias in their identification. Nevertheless, the lack of standard detection strategies used in the tools could be a potential threat. We are aware that there are other tools that can detect many of the code smells analyzed, and their detection strategies could to some extent differ from those used in this study.

Internal Validity. It is possible that some developers were more open about problems than others, and some did not report all the maintenance problems they experienced. This is a common threat whenever qualitative data of the type collected in our study (interviews and think-aloud sessions) is used. Our usage of three independent collection methods, i.e., interviews, direct observation and think-aloud sessions for triangulation[8] purposes, may have reduced this threat.

External Validity. The results are contingent on the contextual properties of the study and the results are mainly valid for maintenance projects in contexts similar to ours. The maintenance work involved medium-sized, Java-based, web-applications, and the programmers completed the tasks individually, i.e., not in teams or pair programming. This last characteristic can affect the applicability of the results in highly collaborative environments.

[8]In the social sciences, triangulation is often used to indicate that more than two methods are used in a study with a view to double (or triple) checking results. This is also called "cross examination."

We do not claim our results fully represent long-term maintenance projects with large tasks, given the size of the tasks and the shorter maintenance period covered in our study. However, the tasks involved may resemble backlog items in a single sprint or iteration within the context of Agile development. To the best of our knowledge, we do not know of other experimental studies of code smells on *in-vivo* maintenance tasks for more than 240 minutes. In this study, we could closely observe the whole maintenance process for a period up to four full-working weeks.

Finally, some code smells may be more important for other types of maintenance tasks and software applications than those conducted in our study. This means that we cannot exclude their importance without significant coefficients in the regression model developed in our study.

5.3 Implications

Results from this study point to ISP Violation as a code smell associated with a wide set of maintenance problems.

We believe that our study is a useful step toward building a more detailed causal chain that addresses how code smells affect maintenance outcomes. Since severity levels of the maintenance problems were not considered, the effect size of the ISP Violation is yet to be investigated in detail. However, the data from the qualitative analysis suggests that ISP Violation does not only frequently lead to maintenance problems, but also that the maintenance problems caused by this code smell may have a substantial impact on maintenance.

Results from this study reveal the inherent complexity of code smell analysis, where the effect of interactions across code smells may have potentially severe consequences on maintenance. Consequently, we believe that further research should investigate code smell combinations besides the study of individual code smells, along with the notion of *inter-smell relations* suggested by Walter and Pietrzak [56].

We also identified instances of entities with zero code smells, which turned out to be extremely problematic due to other design limitations and their coupling with ISP Violators. Consequently, we believe that dependency analysis should be incorporated as an important component in further studies on code smells.

From a practical perspective, our study could contribute to maintainability analyses in industry, where components displaying ISP Violation could be given particular attention to determine their design quality. This study provides a description of types of maintenance problems that ISP Violation may cause, and developers and architects can associate these descriptions with situations they face in their projects to make refactoring/restructuring decisions.

6 Conclusion and Future Work

This research aimed at assessing the capability of twelve code smells to explain maintenance problems. We found strong evidence that ISP Violation constitutes a code smell likely to be associated with problematic files during maintenance. We found that Data Clump was the code smell associated with the lowest probability of problematic files.

Our study constitutes a realistic maintenance project, and we believe that the maintenance problems identified are representative of those experienced in several industry settings. Thus, our results may provide empirical evidence to guide the focus on design aspects that can be used for detecting and avoiding maintenance problems.

We recommend, based on our findings and experience with the current study design, that future studies should include code smell analyses that: 1) include measures indicating the severity of the maintenance problems, and, 2) focus on the interaction effect between code smells, between code smells and other design properties, and between code smells and program size.

7 Appendix A

Table 3.14: Code smells analyzed (Part 1) and detection strategies [34]

Code smells	**Detection strategy**	**Metrics**
Data Class	WOC lower 33 and (NOPA higher 5 or NAM higher 5)	Weight Of Class (WOC) Number Of Public Attributes (NOPA) Number of Accessor Methods (NAM)
Feature Envy	AID higher 4 and AID top 10% and ALD lower 3 and NIC lower 3	Access of Import Data (AID) Access of Local Data (ALD) Number of Import Classes (NIC)
God Class	AOFD top 20% and AOFD higher 4 and WMPC1 higher 20 and TCC lower 33	Access Of Foreign Data (AOFD) Weighted Methods Per Class 1 (WMPC1) Tight Class Cohesion (TCC)
God Method	(LOC top 20% except LOC lower 70) and (NOP higher 4 or NOLV higher 4) and MNOB higher 4	Lines Of Code (LOC) Number Of Parameters (NOP) Number Of Local Variables (NOLV) Max Number Of Branches (MNOB)
ISP violation	(CIW top 20% except CIW lower 10) and AUF lower 50 and COC higher 3	Class Interface Width (CIW) Average Use of Interface (AUF) Clients Of Class (COC)
Misplaced Class	CL lower 0.33 and NOED top 25% and NOED, higher 6 and DD lower 3	Number Of External Dependencies (NOED) Class Locality (CL) Dependency Dispersion (DD)
Refused bequest	AIUR lower 1	Average Inheritance Usage Ratio (AIUR)
Shotgun Surgery	CM top 20% and CM higher 10 and ChC higher 5	Changing Methods (CM) Changing Classes (ChC)

Table 3.15: Code smells analyzed (Part 2) and potential detection heuristics

Code smells	**Detection heuristic**
Data Clump	*Abstract semantic graph* [30] can be analyzed to detect independent groups of fields and methods that appear together in multiple locations
Duplicated code in conditional branches	*Abstract syntax three* [14] can be analyzed to detect conditional statements, and this information can be combined with *clone detection techniques* (e.g., Baxter et al.,[4])
Temporary variable is used for several purposes	Analysis of *abstract semantic graph* can be combined with *semantic analysis* (e.g., Landauer et al.,[24]) to determine the location where temporal variables are defined and determine differences in their context of usage
Use interface instead of implementation	*Abstract semantic graph* can be analyzed to detect castings to implementation classes

8 Appendix B

Table 3.16: Problems report for file(s) contaning ISP Violation (Part 1)

ID	Summary	Type	Reason	Dev	Sys	File
3	Finding task context is difficult for Person and Publication, specially Publication since domain is localized but code is spread	Understanding	Design inconsistency	2	B	Person
12	Variable "Search" used in different contexts, make it difficult to understand	Understanding	Design inconsistency	3	C	DB
15	Bug due to temporal variables repeatedely used for different purposes	Defects from side effects	Inconsistent variables	3	C	DB
20	Difficulties understanding the caching system	Understanding	Pervasiveness	2	B	Person Simula
28	Defects introduced after modifying function create new study (Last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	Person
28	Defects introduced after modifying function create new study (Last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	Simula
28	Defects introduced after modifying function create new study (Last modified/created by info lost during int-string conversion)	Defects from side effects	Large classes	2	B	ObjectStatementImpl
42	Complaints about the size of StudyDAO	Understanding	Large classes	3	D	StudyDAO
43	Debugging problems during task 1, needed to change strategy several times	Modifying	Lack of flexibility	3	D	StudyDAO
45	Difficulties changing the sql queries to adapt to WS	Modifying	Data dependencies	1	D	StudyDAO
66	Difficulties finding the logic for displaying studies	Understanding	Logic spread	4	D	StudyDAO
70	Changes in DB.java triggered a series of errors in jsp files, which are difficult to track	Defects from side effects	Large classes	5	C	DB

Table 3.17: Problems report for file(s) contaning ISP Violation (Part 2)

ID	Summary	Type	Reason	Dev	Sys	File
75	Difficulties understanding the data retrieval (queries) for searching studies	Understanding	Logic spread	4	D	StudyDAO
77	Defects introduced after task 1	Defects from side effects	Large classes	6	A	StudyDatabase
78	Difficulties due to copy-paste error (1 hour ca) from ResultSet variable which was wrong	Defects from side effects	Inconsistent variables	5	C	DB
80	Difficulties understanding business logic due to inconsistencies in the use of methods	Understanding	Design inconsistency	5	C	DB
82	Difficulties while implementing title based search and free text search (considered very difficult)	Modifying	Data dependencies	4	D	StudyDAO
88	Duplicated code in DB statements and connectors	Understanding	Design inconsistency	5	C	DB
91	Problems with change spread in DB (many methods)	Modifying	Large classes	5	C	DB
97	Difficulties due to side-effects from MT3 (isReallyNull) which required adjustment in order to cope with changes in StudyReport	Defects from side effects	Afferent coupling	4	D	Nuller
98	Some corrections were needed due to side-effects from tasks	Defects from side effects	Afferent coupling	4	D	StudyDAO
113	Side-effect from task 1	Defects from side effects	Afferent coupling	4	C	DB
118	Difficulties understanding and using the framework for displaying data	Understanding	Pervasiveness	6	B	Table
137	Side effects (bugs) from code reuse in duplicated logic	Defects from side effects	Inconsistent variables	1	A	StudyDatabase
35	Programmer doesn't understand why StudySortBean is used for study responsibles	Understanding	Semantic inconsistency	1	A	StudySortBean
36	Considered potentially difficult by the developer	Understanding	Design inconsistency	1	A	StudySortBean
47	Difficulties generating the url, related to difficulties with csv strings of ids	Modifying	Implementation	3	D	StudySDTO
50	WebConstants and WebKeys had hardcoded class names and were difficult to refactor, also Frontcontroller_URL had hardcoded paths	Modifying	Implementation	3	D	WebConstants, WebKeysStudy
65	Difficulties changing the hashmap so used caching instead	Modifying	Implementation	1	D	StudyDAO
127	Problems with space characters ("code was strange")	Understanding	Implementation	4	C	DB
26	Defect due to entityKey	Defects from side effects	Internal complexity + coupled to wide interface	2	B	ObjectStatementImpl
27	DB driver problems, and this forced the programmer to do hard casting from long to int	Modifying	Limited design + coupled to wide interface	2	B	ObjectStatementImpl

9 Acknowledgement

The author thanks Gunnar Bergersen for his support in selecting the developers of this study and Hans Christian Benestad for providing technical support in the planning stage of the study. Also, thanks to Bente Anda and Dag Sjøberg for finding the resources needed to conduct this study and for insightful discussions. Thanks to Erik Arisholm for sharing his expertise during the analysis of the data. Finally, special thanks to Magne Jørgensen for his guidance and discussions that led to the paper. This work was partly funded by Simula Research Laboratory and the Research Council of Norway through the projects AGILE, grant no. 179851/I40, and TeamIT, grant no. 193236/I40.

10 References

[1] Marwen Abbes et al. "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension." In: *European Conf. Softw. Maint. and Reeng.* 2011, pp. 181–190.

[2] El Hachemi Alikacem and Houari A. Sahraoui. "A Metric Extraction Framework Based on a High-Level Description Language." In: *Working Conf. Source Code Analysis and Manipulation*. 2009, pp. 159–167.

[3] Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System." In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 407–429.

[4] Ira D. Baxter et al. "Clone detection using abstract syntax trees." In: *Int'l Conf. Softw. Maint.* IEEE Comput. Soc, 1998, pp. 368–377.

[5] Gunnar R. Bergersen and Jan-Eric Gustafsson. "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective." In: *Journal of Individual Differences* 32.4 (2011), pp. 201–209.

[6] Borland. *Borland Together [online] Available at: http://www.borland.com/us/products/together [Accessed 10 May 2012]*. 2012.

[7] William Brown et al. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[8] Peter Coad and Edward Yourdon. *Object-Oriented Design. London,* London: Prentice Hall, 1991.

[9] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. "On the Impact of Design Flaws on Software Defects." In: *Int'l Conf. Quality Softw.* 2010, pp. 23–31.

[10] Ignatios Deligiannis et al. "A controlled experiment investigation of an object-oriented design heuristic for maintainability." In: *Journal of Systems and Software* 72.2 (2004), pp. 129–143.

[11] Ignatios Deligiannis et al. "An empirical investigation of an object-oriented design heuristic for maintainability." In: *Journal of Systems and Software* 65.2 (2003), pp. 127–139.

[12] George Dunteman. *Principal components analysis. Sage university paper series on quantitative applications in the social sciences*. Newbury Park, CA: Sage, 1989.

[13] Edgewall-Software. *Trac [online] Available at: http://trac.edgewall.org [Accessed 10 May 2012]*. 2012.

[14] Gregor Fischer, J. Lusiardi, and J. Wolff von Gudenberg. "Abstract Syntax Trees - and their Role in Model Driven Software Development." In: *Int'l Conf. on Softw. Eng. Advances*. Aug. 2007, pp. 38–38.

[15] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of feature envy bad smells." In: *Int'l Conf. Softw. Maint.* 2007, pp. 519–520.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, p. 395.

[18] Intooitus. *InCode [online] Available at: http://www.intooitus.com/inCode.html [Accessed 10 May 2012]*. 2012.

[19] Elmar Juergens et al. "Do code clones matter?" In: *Int'l Conf. Softw. Eng.* 2009, pp. 485–495.

[20] Henry Kaiser. "An index of factorial simplicity." In: *Psychometrika* 39.1 (1974), pp. 31–36.

[21] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. "An Exploratory Study of the Impact of Code Smells on Software Change-proneness." In: *Working Conf. Reverse Eng.* 2009, pp. 75–84.

[22] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. "Mining Software Repositories with iSPAROL and a Software Evolution Ontology." In: *Int'l Workshop on Mining Software Repositories*. 2007, p. 10.

[23] Miryung Kim et al. “An empirical study of code clone genealogies.” In: *European Softw. Eng. Conf. and ACM SIGSOFT Symposium on Foundations of Softw. Eng.* 2005, pp. 187–196.

[24] Thomas K. Landauer, Peter W Foltz, and Darrell Laham. “An introduction to latent semantic analysis.” In: *Discourse Processes* 25.2-3 (1998), pp. 259–284.

[25] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2005, p. 220.

[26] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2004.

[27] Lucas M. Layman, Laurie A. Williams, and Robert St. Amant. “MimEc.” In: *Proceedings of the 2008 Int’l workshop on cooperative and human aspects of softw. engin.* 2008, pp. 73–76.

[28] Wei Li and Raed Shatnawi. “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.” In: *Journal of Systems and Software* 80.7 (2007), pp. 1120–1128.

[29] Angela Lozano and Michel Wermelinger. “Assessing the effect of clones on changeability.” In: *Int’l Conf. Softw. Maint.* 2008, pp. 227–236.

[30] Evan Mamas and Kostas Kontogiannis. “Towards portable source code representations using XML.” In: *Working Conf. on Reverse Eng.* 2000, pp. 172–182.

[31] Mika V. Mäntylä. “An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement.” In: *Int’l Conf. Softw. Eng.* 2005, pp. 277–286.

[32] Mika V. Mäntylä and Casper Lassenius. “Subjective evaluation of software evolvability using code smells: An empirical study.” In: *Empirical Software Engineering* 11.3 (2006), pp. 395–431.

[33] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. “Bad smells -humans as code critics.” In: *Int’l Conf. Softw. Maint.* 2004, pp. 399–408.

[34] Radu Marinescu. “Measurement and Quality in Object Oriented Design.” Doctoral Thesis. “Politehnica" University of Timisoara, 2002.

[35] Radu Marinescu. “Measurement and quality in object-oriented design.” In: *Int’l Conf. Softw. Maint.* 2005, pp. 701–704.

[36] Radu Marinescu and Daniel Ratiu. “Quantifying the quality of object-oriented design: the factor-strategy model.” In: *Working Conf. Reverse Eng.* 2004, pp. 192–201.

[37] Robert C. Martin. *Agile Software Development, Principles, Patterns and Practice*. Prentice Hall, 2002.

[38] Scott Menard. *Applied logistic regression analysis*. SAGE Publications, 1995.

[39] Naouel Moha. "Detection and correction of design defects in object-oriented designs." In: *ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications*. 2007, pp. 949–950.

[40] Naouel Moha, Yann-Gael Gueheneuc, and Pierre Leduc. "Automatic generation of detection algorithms for design defects." In: *IEEE/ACM Int'l Conf. on Automated Softw. Eng.* 2006, pp. 297–300.

[41] Naouel Moha et al. "A domain analysis to specify design defects and generate detection algorithms." In: *Fundamental Approaches to Softw. Eng.* 4961 (2008), pp. 276–291.

[42] Naouel Moha et al. "DECOR: A Method for the Specification and Detection of Code and Design Smells." In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[43] Akito Monden et al. "Software quality analysis by code clones in industrial legacy software." In: *IEEE Symposium on Software Metrics*. 2002, pp. 87–94.

[44] Raymond Myers. *Classical and modern regression with applications*. Boston, MA. USA: PSW-Kent Publishing Company, 1990.

[45] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." In: *Int'l Conf. Softw. Maint.* 2010, pp. 1–10.

[46] Oracle. *My Sql [online] Available at: http://www.mysql.com [Accessed 10 May 2012]*. 2012.

[47] Plone Foundation. *Plone CMS: Open Source Content Management [online] Available at: http://plone.org [Accessed 10 May 2012]*. 2012.

[48] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. "Clones: What is that smell?" In: *Int'l Conf. Softw. Eng.* 2010, pp. 72–81.

[49] Ananda Rao and Narendar Reddy. "Detecting bad smells in object oriented design using design change propagation probability matrix." In: *Int'l Multiconference of Engineers and Computer Scientists*. 2008, pp. 1001–1007.

[50] Arthur J. Riel. *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley, 1996.

[51] The Apache Software Foundation. *Apache Subversion [online] Available at: http://subversion.apache.org [Accessed 10 May 2012]*. 2012.

[52] The Apache Software Foundation. *Apache Tomcat [online] Available at: http://tomcat.apache.org [Accessed 10 May 2012]*. 2012.

[53] TMate-Sofware. *SVNKit - Subversioning for Java. [online] Available at: http://svnkit.com [Accessed 10 May 2012]*. 2010.

[54] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of type-checking bad smells." In: *European Conf. Softw. Maint. and Reeng.* 2008, pp. 329–331.

[55] Eva Van Emden and Leon Moonen. "Java quality assurance by detecting code smells." In: *Working Conf. Reverse Eng.* 2001, pp. 97–106.

[56] Bartosz Walter and Blazej Pietrzak. "Multi-criteria Detection of Bad Smells in Code with UTA Method 2 Data Sources for Smell Detection." In: *Extreme Programming and Agile Processes in Softw. Eng.* 2005, pp. 154–161.

[57] Aiko Yamashita. *Measuring the outcomes of a maintenance project: Technical details and protocols. (Report no. 2012-11)*. Tech. rep. Oslo: Simula Research Laboratory, 2012, pp. 1–19.

[58] Robert Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.

[59] ZD Soft. *ZD Soft Screen Recorder [online] Available at: http://www.zdsoft.com [Accessed 10 May 2012]*. 2012.

[60] Min Zhang, Tracy Hall, and Nathan Baddoo. "Code Bad Smells: a review of current knowledge." In: *Journal of Software Maintenance and Evolution: Research and Practice* 23.3 (2011), pp. 179–202.

Paper 5

Do Code Smells Reflect Maintainability Aspects Important for Developers? – A Comparative Case Study

Authors: Aiko Yamashita, Leon Moonen

Abstract

Code smells are manifestations of design flaws that can degrade code maintainability. As such, the existence of code smells seems an ideal indicator for maintainability assessments. However, to achieve comprehensive and accurate evaluations based on code smells, we need to know how well they reflect factors affecting maintainability. After identifying which maintainability factors are reflected by code smells and which not, we can use complementary means to assess the factors that are not addressed by smells. This will result in more comprehensive and accurate evaluations of maintainability. This paper reports on an empirical study that investigates the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers. We consider two sources for our analysis: (1) expert-based maintainability assessments of four Java systems before they entered a maintenance project, and (2) observations and interviews with professional developers who maintained these systems during 14 working days and implemented a number of change requests.

Keywords – Maintainability evaluation, Code smells

1 Introduction

Developing strategies for assessing the maintainability of a system is of vital importance, given that significant effort and cost in software projects is due to maintenance [10, 34, 2, 70, 41]. Recently, existence of code smells has been suggested as an approach to evaluate maintainability [50]. Code smells reflect code that can degrade understandability and changeability, and can lead to the introduction of faults [29]. Code smells indicate that there are issues with code quality, such as understandability and changeability, which can lead to the introduction of faults [29]. Beck and Fowler informally describe twenty-two smells and associate them with refactoring strategies to improve the design. Consequently, code smell analysis opens up the possibility for integrating both assessment and improvement in the software maintenance process.

Nevertheless, to achieve accurate maintainability evaluations based on code smells, we need to better understand the "scope" of these indicators, i.e. know their capacity and limitations to reflect software aspects considered important for maintainability. In that way, complementary means can be used to address the factors that are not reflected by code smells. After identifying which maintainability factors are reflected by smell definitions and which not, complementary means such as metrics or expert judgement can be used to assess the factors not addressed by smell definitions. Overall, this will help to achieve more comprehensive and accurate evaluations of maintainability.

Previous studies have investigated the relation between individual code smells and different maintenance characteristics such as effort, change size and defects; but no study has addressed the question of how well code smells can be used for general assessments of maintainability. Anda reports on a number of important maintainability aspects that were extracted from expert-judgement-based maintainability evaluations of four medium-sized Java web applications [6]. She concludes that software measures and expert judgment constitute not opposing, but complementary approaches because they both address different aspects of maintainability.

This paper investigates the extent to which aspects of maintainability that were identified as important by programmers are reflected by code smell definitions. Our analysis is based on an industrial case study where six professional software engineers were hired to maintain the same set of systems that were analyzed in [6]. They were asked to implement a number of change requests over the course of 14 working days. During this time, we conducted daily interviews and one larger wrap-up interview with each of the developers.

We analyze the transcripts of these interviews using a technique called *cross-case synthesis* to compare each developer's perception on the maintainability of the systems and relate it back to code smells. The results from this analysis were compared to the data reported in [6]. This process was repeated for the data that was reported in [6] to

compare and contrast our results with those from expert judgment.

The contributions of this paper are: (1) we complement the findings by Anda [6] by extracting maintainability factors that are important from the software maintainer's perspective; (2) based on manifestations of these factors in an industrial maintenance project, we identify which code smells (or alternative analysis methods) can assess them; and (3) we provide an overview of the capability of current smell definitions to evaluate the overall maintainability of a system.

The remainder of this paper is structured as follows: First we present the theoretical background and related work. Section 3 describes the case study and discusses the earlier results reported by Anda [6]. Section 4 presents and discusses the results from our analysis. Finally, Section 5 summarizes the study findings and presents plans for future work.

2 Theoretical Background and Related Work

2.1 Software Maintainability

Maintainability is one of the software qualities defined by ISO as: "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment and in requirements and functional specifications" [37]. In order to assess maintainability, numerous conceptual surrogates have been defined alongside software quality and measurement frameworks. Recent work on maintainability models can be found in [43, 71], as well as in technical standards defined by ISO [38].

2.2 Maintainability Assessments

There is a wealth of published research on product- and process based approaches for estimating maintenance effort and the related assessments of maintainability in software engineering literature. Examples of product-based approaches that use software metrics to assess maintainability include [67, 25, 59, 65, 26, 84, 78, 5, 8, 35].

Some approaches use hierarchical quality models to relate external quality attributes (such as maintainability) to internal attributes and metrics. This decomposition or Factor-Criteria-Metrics (FCM) approach was first used by McCall and Boehm [62]. Examples of this approach can be found in [68, 39, 40]. Examples of process-centered approaches for maintenance effort estimation can be found in [32, 51]. Many of the process-centered approaches utilize process-related metrics or historical data to generate estimation models. Hybrid approaches combine process and product related factors: Mayrand and Coallier combine capability assessment (ISO/IEC-12207) with static analysis [60], and Rosqvist

combines static analysis with expert judgment [75]. Other examples of "hybrid approaches" can be found in [16, 30, 3, 18]. A literature review on maintenance cost estimation literature can be found in [49], and a systematic review of maintainability evaluation literature in [74].

2.3 Factors Affecting Maintainability

Different code characteristics have been suggested to affect maintainability. Early examples include size (lines of code, LOC) and complexity measures by McCabe [61] and Halstead [33]. Some have attempted to combine them into a single value, called maintainability index [84]. Measures for inheritance, coupling and cohesion were suggested in order to cope with object-oriented program analysis [14].

Pizka and Deissenboeck [71] assert that, even though such metrics may correlate with effort or defects, they have limitations for assessing the overall maintainability of a system. One major limitation is that they only consider properties that can be automatically measured in code, whereas many essential quality issues, such as the usage of appropriate data structures and meaningful documentation, are semantic in nature and cannot be analyzed automatically. Anda [6] reported that software metrics and expert judgment are complementary approaches that address different maintainability factors. Important factors that are not addressed by metrics are: Choice of classes and names, Usage of components, Adequate architecture, Design suited to the problem domain.

2.4 Code Smells

A code smell is a suboptimal design choice that can degrade different aspects of code quality such as understandability and changeability, and could lead to the introduction of faults [29]. Beck and Fowler [29] informally describe 22 code smells and associated them with refactoring strategies to improve the design. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of the system [50]. Code smell analysis allows for integrating both assessment and improvement in the software evolution process. Moreover, code smells constitute software factors that are potentially easier to interpret than traditional OO software measures, since many of the descriptions of code smells in [29] are based on situations that developers face in a daily basis.

Van Emden and Moonen [81] provided the first formalization of code smells and developed an automated code smell detection tool for Java. Mäntylä [55] and Wake [82] proposed two initial taxonomies for code smells. Mäntylä investigated how developers identify and interpret code smells, and how this compares to results from automatic de-

tection tools [54]. Examples of recent approaches for code smell detection can be found in [57, 4, 45, 63]. Automated detection is implemented in commercial tools such as Borland Together[12] and InCode[36], and academic tools such as JDeodorant [27] and iSPARQL [46]. Previous empirical studies have investigated the effects of individual code smells on different maintainability related aspects, such as defects [64, 52, 42, 17, 73], effort [20, 19, 53, 1] and changes [47, 45, 66].

One of the main goals of incorporating code smells to maintainability evaluations is to address a limitation that expert judgment and traditional code metrics have in common: for both approaches, there is no clear path from evaluation to concrete action plans for improvement. As Anda [6] points out, if one asks an expert to identify the areas to modify to improve maintainability, it would be time-consuming and expensive. Likewise, Marinescu [57] and Heitlager [35] point out that a major limitation of metrics is their lack of guidelines to improve their value (and thereby maintainability). Code smells do not suffer from these drawbacks due to their associated refactorings. Moreover, since an increasing number of code smells can be detected automatically, it is appealing to evaluate their capacity to uncover different factors that affect maintainability. The extent to which we understand how well code smells cover these factors determines our ability to address their limitations by alternative means. This will support more comprehensive and cost-effective evaluations of software maintainability.

3 Case Study

3.1 Systems under Analysis

To conduct a longitudinal study of software development, the Simula's Software Engineering Department put out a tender in 2003 for the development of a new web-based information system to keep track of their empirical studies. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all using the same requirements specification. The companies knew, and agreed that the work would be done as part of a research study. More details on the initial project can be found in [7]. The same four functionally equivalent systems are used in our current study. We will refer to them as System A, System B, System C and System D, respectively.

The systems were primarily developed in Java and have similar three-layered architectures, but have considerable differences in their design and implementation. The main functionality of the systems consisted of keeping a record of the empirical studies and related information (e.g., the researcher responsible of the study, participants, data collected and publications resulting from the study). Another major functionality was to

generate a graphical report on the types of studies conducted per year. The systems were deployed over Simula's Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to the database in the CMS, in order to access data related to researchers at Simula as well as information on the publications.

3.2 Software Factors Important to Maintainability

After the systems were developed, two (external) professional software engineers were hired to individually evaluate the maintainability of these systems. The first software engineer had more than 20 years of experience at that time, and the second expert had 10 years of experience. The following is an excerpt of their maintainability assessment based on expert judgment, sorted from highest- to lowest maintainability.

- System A is likely to be the most maintainable, as long as the extensions to the system are not too large.
- System A is likely to be the most maintainable, as long as the extensions to the system are not too large.
- System D shows slightly more potential maintainability problems than System A, especially as some ambitious parts were unfinished.However, System D may be a good choice if the system is to be extended significantly.
- System C was considered difficult to maintain. Small maintenance tasks may be easy, but it is not realistic to think that it could be extended significantly.
- System B is too complex and comprehensive and is likely to be very difficult to maintain. The design would have been more appropriate for a large-scale system.

From the full evaluations, Anda extracted factors that affect maintainability, and concluded that most of them are only addressable by expert judgment, and not by metrics [6]. An overview of these factors is shown in Table 5.1.

3.3 Maintenance Project

In 2008, Simula's CMS was replaced by a new platform called Plone [72] and it was no longer possible to run the systems under this new platform. This gave the opportunity to set up a maintenance study, where the functional similarity of the systems enabled investigating the relation between design aspects and maintainability on cases with very

Table 5.1: Important factors affecting maintainability, as reported in [6]

Factor	**Description**
Appropriate techn. platform	Many maintenance problems are related to undocumented, implicit requirements that surface when a system is moved to a different environment. The use of non-standard third party components poses a challenge to using the components in further development. Developers have to put extra effort in understanding how to use the component. They also have to understand how to replace the component in the future because it may not be maintained or may become unavailable. Related to undocumented, implicit requirements surfacing when a system is moved to a different environment. The use of non-standard third party components poses a challenge to understanding, using and replacing the components in further development.
Coherent naming	Developers should use a consistent naming schema that allows the reader to understand relations between methods and classes. Classes should be easy to identify to facilitate the mapping from domain and requirements to code.
Comments	Comments should be meaningful and size-effective, so they do not influence negatively the readability of the code.
Design suited to problem domain	The complexity of the problem domain must justify the choice for the design. For example, the use of design patterns must be adapted to the project context.
Encapsulation	Since Java methods return only one object, developers often create small "output" container classes as a work-around. This introduces dependencies, such as creating object structures before a method is called, which can lead to maintenance problems.
Inheritance	There must be a balance between adding functionality to a base class or extending it, because it may not be obvious what generic functionality will be required by all or most subclasses. Furthermore, the use of inheritance increases the total number of classes, so therefore should be used with care. If an interface implements several classes, it has the same effect as multiple inheritances, which may lead to confusion and lower maintainability.
Libraries	The use of proprietary libraries may mean lower maintainability, because new developers will need to familiarize themselves with them. Proprietary libraries may also be influenced by the coding style of the developers that created the library, something that may make the code difficult for other developers to understand. The use of proprietary libraries may mean lower maintainability, because new developers will need to familiarize themselves with them.
Simplicity	Size and complexity of a system is critical. It takes longer to identify a specific class when there are many classes. The presence of several classes that are almost empty is a sign of code that may possess low maintainability.
Standard naming conventions	The use of standard naming conventions for packages, classes, methods and variables eases understanding.
Three-layer architecture	A clear separation of concerns between presentation, business and persistence layer is considered good practice. Each layer should remain de-coupled from the layers above it and depend only on more general components in the lower layers.
Use of components	Classes should be organized according to functionality or according to the layer of the code on which they operate.

similar contexts (*e.g.,* identical tasks and programming language), but different designs and implementations. The project was outsourced to two software companies in Eastern Europe at a total cost of 50.000 Euros.

3.4 Maintenance Tasks

Three maintenance tasks were defined, as described in Table 5.2. Two tasks concerned adapting the system to the new platform and a third task concerned the addition of new functionality that users had requested.

Table 5.2: Maintenance tasks

No.	Task	Description
1	Adapting the system to the new Simula CMS	The systems in the past had to retrieve information through a direct connection to a relational database within Simula's domain (information on employees at Simula and publications). Now Simula uses a CMS based on Plone platform, which uses an OO database. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications, as now a String type is used instead. Task 1 consisted of modifying the data retrieval procedure by consuming a set of web services provided by the new Simula CMS in order to access data associated with employees and publications.
2	Authentication through web services	Under the previous CMS, authentication was done through a connection to a remote database and using authentication mechanisms available on that time for Simula Web site. This maintenance task consisted of replacing the existing authentication by calling a web service provided for this purpose.
3	Add new reporting functionality	This functionality provides options for configuring personalized reports, where the user can choose the type of information related to a study to be included in the report, define inclusion criteria based on people responsible for the study, sort the resulting studies according to the date that they were finalized, and group the results according to the type of study. The configuration must be stored in the systems' database and should only be editable by the owner of the report configuration.

3.5 Developers

Six developers were recruited from a pool of 65 participants in a study on programming skill [11] that included maintenance tasks. They were selected based on their availability, English proficiency, and motivation for participating in a research project.

3.6 Activities and Tools

The developers were given an overview of the project and a specification of each maintenance task. When needed, they would discuss the maintenance tasks with the researcher (first author) who was present at the site during the entire project duration. Daily interviews were held where the progress and the issues encountered were tracked. Acceptance tests were conducted once all tasks were completed, and individual open interviews were conducted where the developer was asked upon his/her opinion of the system. The daily

interviews and wrap-up interviews were recorded for further analysis. MyEclipse [31] was used as the development tool, together with MySQL [69] and Apache Tomcat [80]. Defects were registered in Trac [22] (a system similar to Bugzilla), and Subversion or SVN [79] was used as the versioning system.

3.7 Research Methodology

The process to extract the maintainability aspects from the developer interviews followed a chain of evidence strategy as shown in Figure 5.1. It is similar to the process reported by Karlström and Runeson [44].

Observed cases. Each of the six developers individually conducted all three tasks on two systems. This was done to collect more observations for different types of analysis, and gave us a total of 12 cases, 3 observations per system. The assignment of developers to systems was random, with control for equal representation, learning effects (i.e. every system at least once in first round and at least once in second round) and maximizing contrast between the two cases handled by each developer (based on the expert-judgments).

Data collection and summarization. After the developers had finished the maintenance tasks for a one system, individual open-ended interviews (approx. 60 minutes) were held, where the developer was asked to give his/her opinion of the system and underlying reasons for the opinion. The choice for open-ended interviews is based on the rationale that important maintainability aspects should emerge naturally from the interview, and not be influenced by the interviewer. To enable data-triangulation, the daily interviews were transcribed and analyzed to collect data to cross-examine findings from the open-ended interviews. The daily interviews (20-30 minutes) resulted from individual meetings (mostly in the morning) with each developer, to keep track of the progress, and to record any difficulties encountered during the project (ex. Dev: "It took me 3 hours to understand this method..."). All recorded interviews were transcribed and summarized using a tool called Transana [83].

Data analysis. The data was analysed using cross-case synthesis [86] and coding techniques [77], following the guidelines from Edberg [21]. Cross-case synthesis is a technique to summarize and identify tendencies in a multiple case study. Transcripts from the interviews were coded using both open and axial coding, as described in [77]. Open coding is a form of content analysis. Statements from the developers were annotated using labels (codes) that were initially constructed from a logbook that the on-site researcher kept

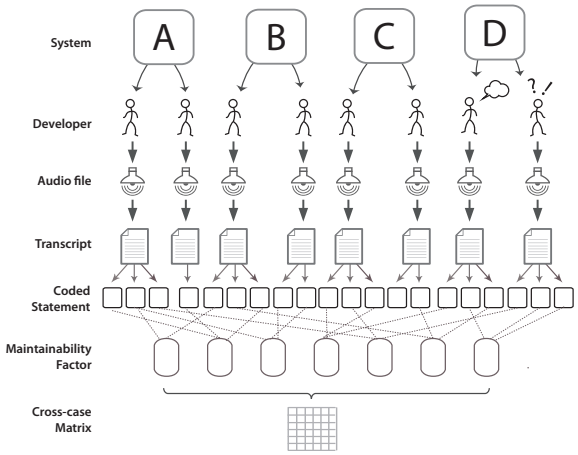

Figure 5.1: Chain of evidence for data summarization and analysis

during the project, and iteratively revised during the annotation process. Example codes are: DB Queries, Size, Bad naming, Lack of rules, Data Access Objects.

For axial coding, the annotated statements were grouped according to the most similar concepts, based on the researchers' observations throughout the project. For more details we refer to our technical report [85]. The grouping was conducted using excel spreadsheets. During this process, we found that many of the categories were similar or identical to the factors reported in [6]. From thereon, factors from [6] were used when applicable. The result was a set of stable and common categories that constitute candidate maintainability aspects. Each aspect was examined for coherence and strength across the cases based on Eisenhardt's recommendations for analyzing within-group similarities coupled with intergroup differences [23]. Some candidate aspects were not replicated across cases, and are therefore not included in the final results.

To analyze the impact of the identified factors, a cross-case matrix was used to compare the factors across cases based on the previous maintainability evaluation and the perception of the maintainers.Finally, each factor was analyzed individually, alongside the statements from the developers that were grouped together, in order to determine the degree to which this factor can be reflected based on the definitions of the twenty-two code smells described by Beck & Fowler [29] and the design principles described by Martin [58].

4 Results

4.1 Comparing Expert Assessment and Developer Impression

From the cross-case synthesis and axial coding, thirtheen maintainability factors emerged. Nine of these confirm earlier findings by Anda [6], and four new factors were identified. Table 5.3 shows the factors that emerged, the number of statements or quotes associated to each factor, and the number of developers who made statements on these factors (this last value enclosed in a parenthesis). Note that the factors "Comments" and "Standard naming conventions" from Anda are not included in our list because only one developer mentioned the first and none mentioned anything related to the second factor.

The last four columns of Table 5.3 show the developers' perception of each factor for every system. The coding is as follows: "M/N Neg" means that N developers talked about that factor in the system and M of them had a negative impression; "M/N Pos" is similar for a positive impression. Note that these are not mutually exclusive, as shown in the table. Moreover, since the perceptions are based on what was mentioned in the interviews, some aspects are not covered for every system, in which case they are marked with "Nm" (not-mentioned). As an example, consider the factor Three-layer architecture: three developers mentioned this for system D, two of them had a positive impression, and one was negative; no developers mentioned this topic for system B (hence "Nm"). Finally, for the factors defined in the work by Anda, the superscript values indicate the degree of matching between the expert evaluation and the developers' impression where 0=no match, 1=medium match, and 2=full match.

If we observe the number of references to maintainability factors, we can distinguish Technical platform, Simplicity and Design consistency as the factors most mentioned by all six developers. The factors with most matches are on the negative impressions on system B, in particular to Technical platform, Design suited to the problem domain, and Simplicity. Systems A and C display a high degree of agreement on positive impression over the Simplicity of the systems. System A displayed the highest rate of disagreements between expert judgment and the maintainers, and System D displayed the highest degree of agreement, both parties considering this system as fairly good.

4.2 Relating Maintainability Factors to Code Smells

Next, for each maintainability factor identified in our analysis, we discuss how it was perceived to affect maintenance, and analyze which code smells relate to it. For factors that cannot be related to code smells, we discuss which alternative methods can be used to evaluate them.

Table 5.3: Cross-Case Matrix of Maintainability Factors and System Evaluations

Factor	**defined in [6]**	**#statements (#dev)**	**A**	**B**	**C**	**D**
Appropriate technical platform	Yes	19 (6)	2/2 Neg^0	3/3 Neg^2	2/2 Neg^2	1/2 Neg, 1/2 Pos^1
Coherent naming	Yes	3 (2)	1/1 Neg^0	Nm	1/1 Neg^0	Nm
Design suited to the problem domain	Yes	6 (3)	Nm	3/3 Neg^1	Nm	Nm
Encapsulation	Yes	3 (3)	2/2 Pos^0	Nm	1/1 Pos^0	Nm
Inheritance	Yes	1 (1)	Nm	1/1 Neg^2	Nm	Nm
(Proprietary) Libraries	Yes	4 (2)	Nm	2/2 Neg^2	Nm	Nm
Simplicity	Yes	21 (6)	3/3 Pos^1	3/3 Neg^2	3/3 Pos^1	1/1 Neg^1
Three-layer architecture	Yes	6 (4)	1/1 Neg^1	Nm	2/2 Neg^2	2/3Pos, 1/3 Neg^2
Use of components	Yes	4 (3)	Nm	1/1 Neg^1	2/2 Neg^2	1/1 Pos^2
Design consistency	No	27 (6)	3/3 Neg	2/3 Pos,1/2 Neg	3/3 Neg	3/3 Pos
Duplicated code	No	2 (2)	2/2 Neg	Nm	Nm	Nm
Initial defects	No	5 (3)	Nm	1/1 Pos	2/2 Neg	2/2 Pos
Logic Spread	No	3 (2)	1/1 Neg	2/2 Neg	Nm	Nm

Table 5.4: Maintainability Factors and their relation to current definitions of Code Smells

Factor	**Covered by code smell**	**Code smells associated**	**Autom. smell detection**	**Alt. evaluation**
Appropriate technical platform	no	NA	no	Expert judgment
Coherent naming	no	NA	no	Semantic analysis, Manual inspection Data Mining Techniques
Design suited to problem domain	partially	Speculative Generality	no	Expert judgment
Encapsulation	partially	Data Clump	partially	Manual inspection
Inheritance	partially	Refused Bequest, Simulation of multiple inheritance	partially	Manual inspection
(Proprietary) Libraries	partially	Wide Subsystem Interface	partially	Expert judgment, Dependency Analysis
Simplicity	partially	God Class, God Method, Lazy Class, Message Chains, Long Parameter List	yes	Expert judgment
Three-layer architecture	no	NA	no	Expert judgment
Use of components	partially	God Class, Misplaced Class	yes	Semantic analysis, Manual inspection
Design consistency	partially	Alternative Classes with Different Interfaces, ISP Violation, Divergent Change, Temporary Field	partially	Semantic analysis, Manual inspection
Duplicated code	yes	Duplicated code, Switch statements	yes	Manual inspection
Initial defects	no	NA	no	Acceptance tests, Regression testing
Logic Spread	partially	Feature Envy, Shotgun Surgery, ISP Violation	yes	Manual inspection, Dependency analysis

Appropriate technical platform. This factor manifested in several forms across the projects. In System B, it appeared in the form of a complex, proprietary Java persistence framework, a particular type of authentication based on Apache Tomcat Realm, and a memory caching mechanism (which became obsolete with the new Simula CMS). Developers claimed that they spent many hours trying to understand each of these mechanisms. One of the experts in [6] stated: *"Many problems with systems maintenance are related to undocumented, implicit requirements that surface when a system is moved to a different environment"*. Here, this was evidenced via two widely used, restrictive interfaces. Both were made under the assumption that identifiers for objects would always be Integers. However, in the new environment, String type object identifiers were needed. Interface replacement was not possible since the implementation was based on primitive types instead of of domain entities.

For systems A and C, this factor manifested in the lack of appropriate persistence mechanisms, resulting in very complex SQL queries embedded in the Java code. Developers saw the integration of Simula's CMS and the SQL queries as one of the biggest challenges. Another example in System C was the log mechanism, which did not stream out the standard error messages generated by Tomcat. As a result, developers were forced to introduce try and catch statements in many segments of the Java and JSP code. The variation in these cases shows that it is difficult for code smells to reflect such situations, instead requiring expert evaluations.

Coherent naming. This factor reflects to how well the *code vocabulary* represents the domain, and how it facilitates the mapping between domain and requirements and code. Examples include System A where a developer did not understand why a class was called "StudySortBean" when its responsibility was to associate a Study to an Employee. In System C, similar situations occurred at the method level (*"One of the most problematic factors was strangely named methods"*). The meaningfulness of a code entity's name cannot be evaluated by code smells, and may require manual inspection and/or semantic analysis.

Design suited to the problem domain. Relates to selecting a design that is adequate for the context of the system. Experts stated: *"...the complexity of the system must justify the chosen solution, and the maintenance staff must be competent to implement a solution in accordance with the design principles"*. System B shows a counter-example, where a complex proprietary persistence library was used instead of a generic one, better suited to small/medium sized information systems. The experts scored System B as somewhat satisfactory, which corresponds to a certain degree to the developers' perceptions. This relates to the Speculative Generality smell. However this factor (and smell) is very difficult

to evaluate via automated code analysis and requires expert judgment and architecture analysis techniques. Examples of architectural evaluation methods can be found in [48, 28], and a set of architecture evaluation criteria is reported in [13].

Encapsulation. Experts in Anda's study concluded that small container classes were used to deliver more than one object as output from methods. They indicated that this introduces dependencies that can lead maintenance problems. In general encapsulation aims to hide the internals of a class to prevent unauthorized access. Developers perceived that Systems A and C had acceptable encapsulation, which resulted in a localized ripple-effect. Code smells such as Data Clumps are indictors of inadequate encapsulation, and they can be complemented with manual inspection.

Inheritance. In System B, multiple interfaces were used to simulate multiple inheritance. However, this practice led to such complex (and dynamic) dependencies between classes that it prompted one of the developers to remove code that he erroneously considered "dead code". After finding out that it wasn't, considerable effort was needed to roll back this change. This case indicates that practices that traditionally are considered ok in OO programming can sometimes cause serious problems. Currently, there are no code smell definitions related to "Simulation of multiple inheritance" but we propose it as a new smell, considering the serious consequences this factor could entail. Given the small/medium size of the other systems, inheritance was not used extensively, and this characteristic did not manifested in the interviews. In addition to "Simulation of multiple inheritance", Refused Bequest ("Subclasses don't want or need everything they inherit") can also be useful to evaluate this factor.

(Proprietary) Libraries. As mentioned before, System B contained a complex proprietary library that transforms logical statements to queries for accessing the database. References to this persistence logic were scattered over the system, forcing the developers to inspect a considerable amount of files in order to understand and use the library, especially for Task 3. The experts in Anda's work indicate that *"the use of libraries may imply a greater amount of code, which in itself is less maintainable. The use of proprietary libraries may imply lower maintainability, because new developers will need to familiarize themselves with them."* This was exactly the case in System B. Although such libraries can be analyzed with static analysis, their maintainability implications depend on how they were used previously and the proportion of the system that relies on them. A design principle violation that relates to this factor is the Wide Subsystem Interface: "A Subsystem Interface consists of classes that are accessible from outside the package they belong to. The flaw refers to the situation where this interface is very wide, which causes

a very tight coupling between the package and the rest of the system" [58]. If Libraries are manifested in the form of packages, this flaw indicates that there is a wide set of clients depending on it. Evaluation approaches are dependency analysis and expert judgment.

Simplicity. Simplicity was considered a very important factor by both experts and maintainers, and clearly distinguishes our four systems. Systems A and C were perceived as simple and fast to get started with, in contrast to System B, which was perceived as extremely complex, in particular due to the number of code elements and the interconnections between them. The large number of classes required time to understand the code at higher level, and to find relevant information. Experts and developers agreed that there was a large proportion of "empty classes" in System B, making it difficult to identify the pertinent ones. A very descriptive remark by one developer was *"I spent more time for understanding than for coding."* Another dimension of this factor is the size of the classes and methods. For all four systems, developers complained about at least two classes "hoarding" the functionality of the system. These were extremely large in comparison to the other classes and displayed high degrees of afferent and efferent coupling dispersion.[1] Referring to this factor, a developer stated: *"Size of methods and classes is important, because I need to remember after reading a method what was it about!"* There are a number of code smells that relate to this factor: God Class and God Method, and Long Parameter List can identify cases as the one previously described. Lazy Class can find the "empty" redirection classes mentioned by the experts. Traditional metrics such as LOC, NOC can be useful to assess this factor as well. Finally, Message Chains can indicate complex, long, data/control flows that typically result from such redirections. Indirect indicators of complex, large classes can manifest via smells that were described previously in the factor Logic Spread, because in the context of information systems, normally large classes "hoard" the functionality of the system, leading to wide dependencies in both directions.

Three-layer architecture. This factor manifested in System C, which had excessive business logic embedded in JSP files. This forced the developers to work with the logic in the JSP files, performing modifications in a "manual" way, as they were deprived from much of the functionality in Eclipse that was only available for Java files. In System D, developers were slightly taken aback due this system having an additional layer inside the business logic layer (enabling the web presentation layer to be replaced by a standalone library in the future), although this was not considered very problematic. Code smells do

[1] Afferent coupling spread or dispersion denoted many elements having dependencies on one element, which is typical of widely used interfaces. Efferent coupling spread or dispersion denotes one element depending on many interfaces.

not reflect this level of abstraction, thus alternative approaches are needed to evaluating this factor. Several approaches have been proposed, most of them relying to a certain degree on human input [15].

Use of components. The organization of classes should be according to functionality, or according to the layer of the code on which they operate. This factor played a role in System C which lacked a clear distinction between business and data layers. The "hoarders" mentioned earlier are orthogonal manifestations of this factor, since classes begin to grow because they cover more functionality than they should. Although semantic aspects of this factor (such as the nature of functionality consistently allocated to corresponding classes) should be evaluated separately, the quantitative perspectives (there should not be classes that do too little or too much) on this factor can be evaluated by code smells such as God Class. In addition, Misplaced Class could be used to identify outliers that are in the wrong layer or package. Given that this factor requires considerable semantic knowledge, it cannot be addressed solely by code smells.

Design consistency. This factor was the one mentioned most by developers during the interviews and was not covered by experts. It refers to the impossibility for developers to oversee the behavior of the system, or the consequences of their changes, because they were constantly facing contradictory or inconsistent evidence. The inconsistencies in System C manifested both at the variable level (*"I had troubles with bugs in DB class, from mistakes in using different variables"*) and at the design level (*"Design in C was not consistent, similar functionality was implemented in different ways"*).

Confusing elements also occurred in system A, where the data or functionality allocation was not semantically coherent, nor consistent (*"Data access objects were not only data access objects, they were doing a lot of other things"*). This resulted in false assumptions about the system's behavior, and developers would get confused when they found evidence that contradicted earlier assumptions. Alternatively, when the false assumptions were not confuted, they led to the introduction of faults (*"The biggest challenge was to make sure changes wouldn't break the system; because things were not always what they seemed to be"*). A contrasting example was system D, which was perceived as very consistent by all three developers who worked with it. Quotes from the developers illustrate this:

- "It was about applying the same pattern of changes for similar classes"
- "There were no surprises, if I change the class X, I could follow the same strategy to change class Y"
- "If something breaks, in system D was easier to trace the fault"
- "It was relatively easier to understand system D compared to A"

Developers working with system D indicated that having a consistent schema for variables, functionality and classes was a clear advantage for code understanding, information searching, impact analysis and debugging. They indicated that normally, they would learn a certain solution pattern by examining segments of the system, and expect this pattern to hold within the system. Most of the observed cases were of semantic nature, which were combined with structural-related factors, thus evaluating this factor constitutes a rather challenging task. We suggest a couple of code smells that can potentially help to identify design inconsistencies, as described in Table 5.5.

Table 5.5: Code Smells related to the identification of design inconsistencies

Code Smell	**Description**
Alternative Classes with Different Interfaces	Classes that mostly do the same things, but have methods with different signatures
Divergent Change	One class is commonly changed in different ways for different reasons
Temporary Field used for several purposes	Sometimes temporary variables are used across different contexts of usage within a method or a class.

In addition, ISP Violation can be an indirect indicator of inconsistency. Martin [58] states that: "Many client specific interfaces are better than one general purpose interface". When "fat interfaces" start acquiring more and more responsibilities and start getting a wider spectrum of dissimilar clients, this could affect analyzability and changeability. The presence of ISP Violation may not imply the presence of design inconsistencies, but one can reasonably assume that chances of finding inconsistencies in an wide interface is higher than interfaces that are used only by a small set of clients. Alternative options to evaluate it can be semantic analysis, and manual inspection.

Duplicated code. The maintainability of a system can be negatively affected by including blocks of code that are very similar to each other (code clones). In Anda's work, the experts include this as an aspect of Simplicity [6]. In our study, we identified cases where Simplicity and Duplicated code manifested as separate factors, which is why we add this factor as an independent one. The developers stated that System A contained many copy-paste related ripple-effects and they considered duplication as one of the biggest difficulties. Duplicated code has attracted considerable attention from the research community and various approaches exist for detection, removal and evolution in the presence of clones. An additional related smell is Switch Statements where conditionals depending of type lead to duplication.

Initial defects. A factor affecting maintainability that was not mentioned by the experts in Anda's work is the amount of defects in the system. In the case of System C, developers unanimously complained about how many defects the system contained *at the beginning of the maintenance phase*. They claimed that they spent a lot of time correcting defects before they could actually complete the tasks. Systems D and B were perceived as not having many initial defects. Although some work has been done for predicting the defect density at class level using code smells [52], in general this factor needs to be evaluated by other means, such as a set of regression tests, acceptance tests, etc.

Logic Spread. Maintainers of system B mentioned that it contained a set of classes whose methods made many calls to methods or variables in other classes, and that they were forced to examine all the files called by these methods, resulting in considerable delays. A developer mentioned: *"It was difficult in B to find the places to perform the changes because of the logic spread"*. Although the factor Simplicity is related to this factor, Simplicity only covers the amount of elements in a system, and not the number of interactions between them. Logic spread has been addressed early by metrics like afferent and efferent coupling. Smells such as Feature Envy indicate efferent coupling dispersion. Other smells such as Shotgun Surgery and Interface Segregation Principle Violation focus on afferent coupling dispersion, consequently they can be useful to evaluate this factor as well. Moreover, it is related to the scattering and tangling that is typically associated with implementing *crosscutting concerns* in non-aspect-oriented languages [56].

5 Discussion

5.1 Comparison of Factors Across Sources

Thirteen factors were identified during the maintenance project, nine of them coinciding between experts and maintainers. This supports the findings in [6]. Yet, some factors were mentioned more often than others by the developers. For instance, factors such as *Standard naming conventions* and *Comments* did not play such an important role for developers. Discrepancies can be due to the fact that experts may lack enough contextual information at the time of the evaluation to weigh the impact or importance of a given factor accordingly. Developers in the other hand are conditioned by their maintenance experience, so factors such as the nature of the task may play an important role over what is perceived as the most relevant of the software factors.

Conversely, all developers who participated in the project unanimously mentioned factors such as Appropriate technical platform and Simplicity. This may indicate that such factors should be given more attention, since they may play an important role regardless

of the maintenance context. This result also suggests that to achieve more accurate maintainability evaluations based on expert judgment, the usage of enough contextual detail may be required to enable experts the priorization of certain factors over others. One example of such approach is the usage of "maintenance scenarios" proposed by Bengtsson and Bosch [9].

Some factors identified during the project were not reported previously, and they can complement the findings from [6]. Factors such as *Design consistency* and *Logic Spread* were perceived as very influential by the maintainers, but were not mentioned by the experts. This can be due to the fact that system maintainability evaluations based on expert-judgment focus on factors at higher abstraction levels, as opposed more fine-grained factors observed by developers. Developers will necessarily capture different factors than experts do, because they are the ones having to dive into the code and suffer the consequences of different design flaws. In addition, most expert evaluations are time-constrained, which would not allow the experts to go into many details. The multiplicity of perspectives observed in this study reflects on previous ideas about the need of diversity in evaluation approaches in order to attain more complete pictures of maintainability.

5.2 Scope/Capability of Code Smells

Situations described by the developers during the interview provided a clear-cut outlook on the difficulty of using code smell definitions for analyzing certain maintainability factors. Such was the case for factors as Appropriate technical platform, Coherent naming, Design suited to the problem domain, Initial defects and Architecture. These factors would need additional techniques to be assessed. Yet, it is worth noting that factors related to code smells include both factors addressed by traditional code metrics as well as factors mainly addressed by expert judgment. For instance, some code smells can support the analysis of factors such as Encapsulation and Use of components (which according to [6] are not captured by software metrics). Factors as Simplicity, which are traditionally addressed by static analysis means, are closely related to the code smells suggested in Table 5.4.

Moreover, several detection techniques for these smells are based on size-related software metrics. Logic spread factor also is largely related to the notion of coupling and cohesion initially described in [24]. These results hint at the potential of code smells to cover a more heterogeneous spectrum of factors than software metrics and expert judgment individually.

An interesting factor identified throughout the study was Design consistency, which according to developers played a major role during different maintenance activities (e.g., understanding code, identifying the areas to modify, coding, debugging). This factor was interpreted in a broad sense, crosscutting abstraction levels (e.g., consistency at variable,

method, class level) and was not limited to the "naming" of elements. Despite this rather broad and inclusive definition, we see great potential for a number of code smells that each can help to identify a certain subset of inconsistencies.

For the identified maintainability factors, we find that eight of them are addressable by current code smell definitions. However, in most cases these code smells would need to be complemented with alternative approaches such as semantic analysis and manual inspection. The suggestions on code smells presented in this work were derived theoretically, based on definitions of code smells available in the literature, and as such, they should be treated as suggestions or starting points for further empirical studies to validate their usefulness. We make no claims concerning the degree or descriptive richness of a smell in relation to a maintainability factor, since that would fall out of the scope of this work. Moreover, some code smells are not detectable via automated means, so even though they reflect certain maintenance factors, other means are needed to assess these factors, for which approaches or techniques for determining their presence in the code would be needed to evaluate their descriptive richness/usefulness. Finally, these results are contingent on the nature of the tasks and characteristics of the maintenance project, for which are suggested as a preliminary set of factors. Replications of this study in different industrial contexts are needed in order to extend and support our findings.

5.3 Threats to Validity

Following a grounded theory approach, the most important quality aspects of our findings are their *fit* and *relevance* [76]. For the fit perspective we argue that the different software maintainability factors were consistently assigned to different categories, most of them common knowledge in the software engineering research community. With respect to relevance, we argue that most factors were relevant to both the experts who evaluated the systems, and the developers who maintained it. We addressed the construct validity threat with data-triangulation and investigator triangulation. With respect to internal validity, we argue that most of our results are descriptive, although it relies on the interpretation of the researcher who carried out the interviews. Transcripts from the daily interviews compensate for any potentially concealed issues that developers did not want to discuss during the open-ended interviews. With respect to external validity, the results are contingent to several contextual factors, such as the systems (i.e., medium-size Java web information systems), the nature and size of the tasks, and the project modality (i.e., solo-projects). Finally, one can argue on the representativeness of the maintenance tasks carried out, although all of them were based on real needs and their duration and complexity are representative of real-life projects.

6 Conclusion and Future Work

By understanding the capability and limitations of different evaluation approaches to address different factors influencing software maintainability, we can achieve better overall evaluations of maintainability. Code smells can provide insight on different maintainability factors which can be improved via refactoring. However, some factors are not reflected by code smells and require alternative approaches to evaluate them.

This paper describes a set of factors that were identified as important for maintainability by experts who evaluated the four Java systems in our study [6], and by six developers who maintained those systems for 14 days while implementing a number of change requests. Through our analysis, we identified some new factors not reported in previous work [6], which were perceived as important by the developers.

Based on the explanations from the developers, we found that some of the factors can potentially be evaluated (at least partially) by using some of the current code smell definitions. The contributions of this paper are three-fold:
(1) We confirm and complements the findings by Anda [6], by extracting maintainability factors that are important from the software maintainer's perspective, and (2) Based on the manifestations of these factors during an industrial maintenance project, we identify which code smells (or alternative analysis methods) can evaluate them, and (3) We provide an overview of the capability of code smell definitions to evaluate the overall maintainability of a system. Given the fact that there are many important factors not addressable by static analysis, and that not all code smells are actually automatically detectable, we agree with the statements from Anda [6] and Pizka & Deissenboeck [71] that there is a need for combining different approaches in order to achieve more complete, and accurate evaluations of overall maintainability of a system.

Future work includes detailed analysis of problems reported by developers during the maintenance project and associate them with code smells detected automatically within the code. This can to provide a quantitative perspective in relation to the coverage-level or capability of code smells to uncover problematic code.

7 Appendix A

Table 5.6: Excerpt of statements associated to different maintainability factors

System	Statement	Factor
A	"A is a simple system and it was fast to start working with"	Simplicity
	"Big challenge to make sure changes wouldnâĂŹt break the system; things were not always as they seemed to be"	Design consistency
	"It has too much freedom or is too arbitrary with respect to the design"	Design consistency
	"It was quite fast to learn and understand because it was not too complex"	Simplicity
	"Data access objects were not only data objects, they were doing a lot of other things"	Design consistency
	"Bugs were proportional to the changes in a class, but they will stay within the class"	Encapsulation
B	"Spent long time extending it because there were so many manual changes"	Technical platform
	"Spent long time learning how the system worked"	Simplicity
	"There was a serious lack of flexibility with the framework for generating the reports"	Technical platform
	"I was constantly changing the persistence layer, requiring a lot of coding"	Technical platform
	"The interface was too limited"	Technical platform
	"Spent quite a lot of time in understanding a library for generating SQL queries"	Libraries
	"It was a lot of rework, had to go back and change things because of the framework"	Technical platform
	"The main problem in B were the identifiers for people and publication"	Technical platform
	"Task 1 was easier in A than in B because of the construction of queries in B"	Technical platform
C	"The system is maintainable because its small"	Simplicity
	"Had to rework the queries in the main class several times"	Technical platform
	"System was full of bugs"	Initial defects
	"The bugs will only break one screen"	Encapsulation
	"C is straightforward, easy to start working with"	Simplicity
	"I had troubles with bugs in DB class, and mistakes in using different variables"	Design consistency
	"In C one big class sometimes made it hard to find the right method"	Use of components
	"A messy system but not that complicated, so easy to learn"	Simplicity
	"Everything had to be done from scratch"	Technical platform
	"Most problematic java element: lack of comments and strangely named methods"	Coherent naming
D	"Some extra work for the additional layer with Handlers, but in general it looked good"	Architecture
	"System is balanced, no classes or methods that do too much"	Use of Components
	If I change the class X, I could follow the same strategy to change class Y	Design consistency
	"Spent 10 times more learning this system than the previous, small system"	Simplicity
	"There were not many bugs"	Initial defects

8 References

[1] Marwen Abbes et al. “An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension.” In: *European Conf. Softw. Maint. and Reeng.* 2011, pp. 181–190.

[2] Alain Abran and Hong Nguyenkim. “Analysis of maintenance work categories through measurement.” In: *Int’l Conf. Softw. Maint.* 1991, pp. 104–113.

[3] Yunsik Ahn et al. “The software maintenance project effort estimation model based on function points.” In: *Journal of Software Maintenance* 15.2 (2003), pp. 71–85.

[4] El Hachemi Alikacem and Houari A. Sahraoui. “A Metric Extraction Framework Based on a High-Level Description Language.” In: *Working Conf. Source Code Analysis and Manipulation*. 2009, pp. 159–167.

[5] Mohammed Alshayeb and Li Wei. “An empirical validation of object-oriented metrics in two different iterative software processes.” In: *IEEE Transactions on Software Engineering* 29.11 (2003), pp. 1043–1049.

[6] Bente C. D. Anda. “Assessing Software System Maintainability using Structural Measures and Expert Assessments.” In: *Int’l Conf. Softw. Maint.* 2007, pp. 204–213.

[7] Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. “Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System.” In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 407–429.

[8] Hans Benestad, Bente Anda, and Erik Arisholm. “Assessing Software Product Maintainability Based on Class-Level Structural Measures.” In: *Product-Focused Softw. Process Improvement*. 2006, pp. 94–111.

[9] Perolof Bengtsson and J. Bosch. “Architecture level prediction of software maintenance.” In: *European Conf. Softw. Maint. and Reeng.* (1999), pp. 139–147.

[10] Keith H. Bennett. “An introduction to software maintenance.” In: *Journal of Information and Software Technology* 12.4 (1990), pp. 257–264.

[11] Gunnar R. Bergersen and Jan-Eric Gustafsson. “Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective.” In: *Journal of Individual Differences* 32.4 (2011), pp. 201–209.

[12] Borland. *Borland Together [online] Available at: http://www.borland.com/us/products/together [Accessed 10 May 2012]*. 2012.

[13] Eric Bouwers, Joost Visser, and Arie van Deursen. “Criteria for the evaluation of implemented architectures.” In: *Int’l Conf. Softw. Maint.* Sept. 2009, pp. 73–82.

[14] Shyam R. Chidamber and Chris F. Kemerer. “A metrics suite for object oriented design.” In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[15] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. Addison-Wesley, 2001.

[16] Vianney Cote and Denis St.Pierre. “A model for estimating perfective software maintenance projects.” In: *Int’l Conf. Softw. Maint.* 1990, pp. 328–334.

[17] Marco D’Ambros, Alberto Bacchelli, and Michele Lanza. “On the Impact of Design Flaws on Software Defects.” In: *Int’l Conf. Quality Softw.* 2010, pp. 23–31.

[18] Andrea De Lucia, Eugenio Pompella, and Silvio Stefanucci. “Assessing effort estimation models for corrective maintenance through empirical studies.” In: *Inf. and Softw. Tech.* 47.1 (2005), pp. 3–15.

[19] Ignatios Deligiannis et al. “A controlled experiment investigation of an object-oriented design heuristic for maintainability.” In: *Journal of Systems and Software* 72.2 (2004), pp. 129–143.

[20] Ignatios Deligiannis et al. “An empirical investigation of an object-oriented design heuristic for maintainability.” In: *Journal of Systems and Software* 65.2 (2003), pp. 127–139.

[21] Dana Edberg and Lorne Olfman. “Organizational learning through the process of enhancing information systems.” In: *Int’l Conf. on System Sciences*. 2001, p. 10.

[22] Edgewall-Software. *Trac [online] Available at: http://trac.edgewall.org [Accessed 10 May 2012]*. 2012.

[23] Kathleen M. Eisenhardt. “Building Theories from Case Study Research.” In: *Academy of Management Review* 14.4 (1989).

[24] N. Fenton. “Software measurement: A necessary scientific basis.” In: *IEEE Transactions on Software Engineering* 20.3 (1994).

[25] Elaine H. Ferneley. “Design metrics as an aid to software maintenance: an empirical study.” In: *Journal of Software Maintenance* 11.1 (1999), pp. 55–72.

[26] Fabrizio Fioravanti. “Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems.” In: *IEEE Transactions on Software Engineering* 27 (2001), pp. 1062–1084.

[27] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and removal of feature envy bad smells." In: *Int'l Conf. Softw. Maint.* 2007, pp. 519–520.

[28] Eelke Folmer and Jan Bosch. "A pattern framework for software quality assessment and tradeoff analysis." In: *International Journal of Software Engineering and Knowledge Engingeering* 17.04 (2007), pp. 515–538.

[29] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[30] Alfonso Fuggetta et al. "Applying GQM in an industrial software factory." In: *ACM Trans. Softw. Eng. Meth.* 7.4 (1998), pp. 411–448.

[31] Genuitec. *My Eclipse [online] Available at: http://www.myeclipseide.com [Accessed 10 May 2012]*. 2012.

[32] Juan Carlos Granja-Alvarez and Manuel José Barranco-García. "A Method for Estimating Maintenance Cost in a Software Project." In: *Journal of Software Maintenance* 9.3 (1997), pp. 161–175.

[33] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier, 1977, p. 128.

[34] Warren Harrison and Curtis Cook. "Insights on improving the maintenance process through software measurement." In: *Int'l Conf. Softw. Maint.* 1990, pp. 37–45.

[35] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability." In: *Int'l Conf. Quality of Inf. and Comm. Techn.* 2007, pp. 30–39.

[36] Intooitus. *InCode [online] Available at: http://www.intooitus.com/inCode.html [Accessed 10 May 2012]*. 2012.

[37] ISO/IEC. *International Standard ISO/IEC 9126*. Tech. rep. Geneva: International Organization for Standardization, 1991.

[38] ISO/IEC. *ISO/IEC Technical Report 19759:2005*. 2005.

[39] ISO/IEC. *Software Engineering - Product quality - Part 1: Quality model*. Tech. rep. 2001.

[40] Bansiya Jagdish and G Davis Carl. "A Hierarchical Model for Object-Oriented Design Quality Assessment." In: *IEEE Transactions on Software Engineering* 28.1 (2002), pp. 4–17.

[41] Capers Jones. *Estimating software costs*. McGraw-Hill, 1998, p. 725.

[42] Elmar Juergens et al. “Do code clones matter?” In: *Int’l Conf. Softw. Eng.* 2009, pp. 485–495.

[43] Mira Kajko-Mattsso et al. “A Model of Maintainability - Suggestion for Future Research.” In: *Software Engineering Research and Practice*. CSREA Press, 2006, pp. 436–441.

[44] Daniel Karlström and Per Runeson. “Integrating agile software development into stage-gate managed product development.” In: *Empirical Software Engineering* 11.2 (2006), pp. 203–225.

[45] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. “An Exploratory Study of the Impact of Code Smells on Software Change-proneness.” In: *Working Conf. Reverse Eng.* 2009, pp. 75–84.

[46] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. “Mining Software Repositories with iSPAROL and a Software Evolution Ontology.” In: *Int’l Workshop on Mining Software Repositories*. 2007, p. 10.

[47] Miryung Kim et al. “An empirical study of code clone genealogies.” In: *European Softw. Eng. Conf. and ACM SIGSOFT Symposium on Foundations of Softw. Eng.* 2005, pp. 187–196.

[48] Jens Knodel et al. “Static evaluation of software architectures.” In: *European Conf. Softw. Maint. and Reengineering*. Mar. 2006, pp. 279–294.

[49] Jussi Koskinen and Tero Tilus. *Software maintenance cost estimation and modernization support*. Tech. rep. Information Technology Research Institute, University of Jyvaskyla, 2003, p. 62.

[50] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2005, p. 220.

[51] Hareton K. N. Leung. “Estimating Maintenance Effort by Analogy.” In: *Empirical Software Engineering* 7.2 (2002), pp. 157–175.

[52] Wei Li and Raed Shatnawi. “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.” In: *Journal of Systems and Software* 80.7 (2007), pp. 1120–1128.

[53] Angela Lozano and Michel Wermelinger. “Assessing the effect of clones on changeability.” In: *Int’l Conf. Softw. Maint.* 2008, pp. 227–236.

[54] Mika Mäntylä. “Software Evolvability - Empirically Discovered Evolvability Issues and Human Evaluations.” PhD Thesis. Helsinki University of Technology, 2009.

[55] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. “A taxonomy and an initial empirical study of bad smells in code.” In: *Int’l Conf. Softw. Maint.* 2003, pp. 381–384.

[56] Marius Marin, Arie Van Deursen, and Leon Moonen. “Identifying Crosscutting Concerns Using Fan-In Analysis.” In: *ACM Trans. Softw. Eng. Meth.* 17.1 (Dec. 2007).

[57] Radu Marinescu. “Measurement and quality in object-oriented design.” In: *Int’l Conf. Softw. Maint.* 2005, pp. 701–704.

[58] Robert C. Martin. *Agile Software Development, Principles, Patterns and Practice.* Prentice Hall, 2002.

[59] Karl S. Mathias et al. “The role of software measures and metrics in studies of program comprehension.” In: *ACM Southeast regional Conf.* 1999, p. 13.

[60] Jean Mayrand and Francois Coallier. “System acquisition based on software product assessment.” In: *Int’l Conf. Softw. Eng.* 1996, pp. 210–219.

[61] Thomas J. McCabe. “A Complexity Measure.” In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320.

[62] Jim A. McCall, Paul G. Richards, and Gene F. Walters. *Factors in Software Quality.* Vol. I. NTIS, 1977.

[63] Naouel Moha et al. “DECOR: A Method for the Specification and Detection of Code and Design Smells.” In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[64] Akito Monden et al. “Software quality analysis by code clones in industrial legacy software.” In: *IEEE Symposium on Software Metrics.* 2002, pp. 87–94.

[65] S. Muthanna et al. “A maintainability model for industrial software systems using design level metrics.” In: *Working Conf. Reverse Eng.* 2000, pp. 248–256.

[66] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. “Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems.” In: *Int’l Conf. Softw. Maint.* 2010, pp. 1–10.

[67] Paul Oman and Jack Hagemeister. “Construction and testing of polynomials predicting software maintainability.” In: *Journal of Systems and Software* 24.3 (1994), pp. 251–266.

[68] Paul Oman and Jack Hagemeister. “Metrics for assessing a software system’s maintainability.” In: *Int’l Conf. Softw. Maint.* 1992, pp. 337–344.

[69] Oracle. *My Sql [online] Available at: http://www.mysql.com [Accessed 10 May 2012]*. 2012.

[70] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, 1996, p. 384.

[71] Markus Pizka and Florian Deissenboeck. "How to effectively define and measure maintainability." In: *Softw. Measurement European Forum*. 2007.

[72] Plone Foundation. *Plone CMS: Open Source Content Management [online] Available at: http://plone.org [Accessed 10 May 2012]*. 2012.

[73] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. "Clones: What is that smell?" In: *Int'l Conf. Softw. Eng.* 2010, pp. 72–81.

[74] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. "A systematic review of software maintainability prediction and metrics." In: *Int'l Conf. Softw. Eng. and Measurement*. 2009, pp. 367–377.

[75] Tony Rosqvist, Mika Koskela, and Hannu Harju. "Software Quality Evaluation Based on Expert Judgement." In: *Softw. Quality Control* 11.1 (2003), pp. 39–55.

[76] James Shanteau. "Competence in experts: The role of task characteristics." In: *Organizational Behavior and Human Decision Processes* 53.2 (1992), pp. 252–266.

[77] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[78] Giancarlo Succi et al. "Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics." In: *Journal of Systems and Software* 65.1 (2003), pp. 1–12.

[79] The Apache Software Foundation. *Apache Subversion [online] Available at: http://subversion.apache.org [Accessed 10 May 2012]*. 2012.

[80] The Apache Software Foundation. *Apache Tomcat [online] Available at: http://tomcat.apache.org [Accessed 10 May 2012]*. 2012.

[81] Eva Van Emden and Leon Moonen. "Java quality assurance by detecting code smells." In: *Working Conf. Reverse Eng.* 2001, pp. 97–106.

[82] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003, p. 235.

[83] WCER. *Transana [online] Available at: http://www.transana.org [Accessed 10 May 2012]*. 2012.

[84] Kurt Dean Welker. "Software Maintainability Index Revisited." In: *CrossTalk – Journal of Defense Software Engineering* (2001).

[85] Aiko Yamashita and Leon Moonen. *Do code smells reflect important maintainability aspects?* Technical Report (2012-10). Simula Research Laboratory, 2012.

[86] Robert Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.

Paper 6

Using Concept Mapping for Maintainability Assessments

Authors: Aiko Yamashita, Bente Anda, Dag Sjøberg, Hans Christian Benestad, Per Einar Arnstad, Leon Moonen

Abstract

Many engineering are difficult to define and measure. One example is software maintainability, which has been the subject of considerable research and is believed to be a critical determinant of total software costs. We propose using concept mapping, a well-grounded method used in social research, to operationalize the concept of software maintainability according to a given goal and perspective in a concrete setting. We apply this method to describe four systems that were developed as part of an industrial multiple-case study. The outcome is a conceptual map that displays an arrangement of maintainability constructs, their interrelations, and corresponding measures. Our experience is that concept mapping (1) provides a structured way of combining static code analysis and expert judgment; (2) helps in the tailoring of the choice of measures to a particular system context; and (3) supports the mapping between software measures and aspects of software maintainability. As such, it constitutes a useful addition to existing frameworks for evaluating quality, such as ISO/IEC 9126 and GQM, and tools for static measurement of software code. Overall, concept mapping provides a systematic, structured, and repeatable method for developing constructs and measures, not only of maintainability, but also of software engineering phenomena in general.

1 Introduction

The maintainability of a software system is usually a critical determinant of software costs, yet it is very difficult to evaluate. A primary difficulty comes from the fact that software maintenance involves dealing with many factors, ranging from technological features to human dynamics and cognition, all of them comprising many different and complex settings. These contextual factors limit the generalization of the findings of individual studies. Hence, it is important to determine why the contextual factors play such an intrinsic role in maintenance. We consider, in line with Pizka and Deisenböck [35], that maintainability is not solely a property of a system, but touches on three different dimensions: (a) the people performing software maintenance, (b) the technical properties of the system under consideration, and (c) the maintenance goals and tasks.

With respect to dimensions (a) and (c), it is known that notions such as opportunistic comprehension [25] and information requirement [33] seem to explain some aspects of how large commercial software systems are understood and maintained. Such notions suggest that the skills and experience of the developer and the nature of the maintenance task drive the process of comprehending the system during maintenance. Yet procedures for maintainability assessment have paid little attention to how the programmer or maintainer understands the system or what their information needs are to perform a particular maintenance task. Many of the approaches to assessment tend to isolate the system from its environment by focusing only on the system's technical properties, which limits the scope and accuracy of these approaches. With respect to dimension (b), most work suggests describing the technical properties of a system by quantifiable means, such as software measures, and connecting them afterwards to higher-level quality attributes. ISO 9126 [16] is an example of such an approach. Nowadays, many technical properties of the system can be measured automatically. Still, the central difficulty is to establish relationships between the quantifiable measures and the quality attributes, such as maintainability. One problem is that the effect of the technical properties on maintainability depends on the context of the system. Consequently, context-specific models are needed.

Given that the nature of the maintenance goals and tasks plays an important role during software maintenance, the constructs representing software maintainability should, to some extent, be goal-driven. From a practical perspective, we should ask ourselves: "Is the system good enough for what we plan to do with it?" We conjecture that many inaccuracies generated by, and issues regarding construct validity with, software quality models are due to their rigid nature, which means that they cannot adapt to the specifics of the organizational context. That being so, there is a need to provide methodological support for building, adapting, and validating such models for a given context or setting. In this paper, we propose to use *concept mapping* [42] as a method to incorporate

contextual information in the operationalization of software engineering constructs. We show how concept mapping can be used to systematically derive measures that can be analyzed and interpreted in order to assess the maintainability of a system. We suggest using expert judgment in the concept mapping process for deriving the contextual information. Anda [1] suggested that combining expert knowledge with static code analysis is a viable approach to evaluation, because these strategies address different attributes and dimensions of a system. Static code analysis enables the use of empirical evidence and existing models of software quality. Conversely, expert judgment incorporates contextual and cognitive factors into the analysis, thus supporting more realistic interpretations of the technical properties of the system.

The remainder of this paper is organized as follows. Section 2 describes the theoretical foundation of concept mapping. Section 3 describes how we used this method to develop a map of maintainability constructs according to a given perspective. Section 4 discusses the proposed method. Section 5 discusses some challenges that need to be met when using our approach. Section 6 summarizes the method and offers directions for further research.

2 Concept Mapping

We now outline the theoretical foundations of concept mapping and, following Trochim [40], describe the steps required to implement it.

2.1 Programme Evaluation

Concept mapping is a method commonly used in social research to plan and evaluate programmes [38]. Programme evaluation is a formalized approach for studying the goals, processes, and results of public and private projects, governmental development policies, and programmes. Programme evaluation and software assessments face similar challenges. For instance, programme evaluation models the aspects of the project, policy, or programme under evaluation. Similarly, in software engineering, a quality model of a given system should be defined on the basis of relevant socio-technical properties, such as, characteristics of the developers and privacy policies. While traditional software assessments focus mainly on the technical factors, we see great potential for approaches from social research to incorporate other factors as well. Concept mapping is one example of such an approach.

2.2 Concept Mapping Definition

To conceptualize a problem means to organize one's ideas about the topic of the problem such that pertinent entities, and the relations between them, are specified. This process first identifies the pertinent entities by acquiring input from several relevant theories or groups of people involved in the programme. Second, it determines the underlying relationships between the identified entities by using multidimensional scaling and cluster analysis. Concept mapping is one type of structured conceptualization. It consists of a sequence of concrete operational steps, which yields a conceptual representation [40] of the element(s) under analysis. Conceptualization processes are considered valuable for programme evaluation because they help to gather information about the actors in the project (various stakeholders, authority groups, etc.) from a variety of perspectives. Information about the various categories of participants represents one kind of context information of the programme.

2.3 The Concept Mapping Process

As described by Trochim [42], a concept mapping process has six main steps: preparation of the process, generation of statements (in which the conceptual domain is defined), structuring of statements (the relationships between the domain entities are established), representation of statements (by textual, pictorial, or mathematical means), interpretation of concept maps, and utilization of concept maps. An initiator requests the concept mapping process, and a facilitator leads the group of participants through the various steps.

Preparation

The main goal of the preparation stage is to select the participants and define the focus of the conceptualization, as a precursor to generating statements about the conceptual domain. Trochim suggests selecting about 10 to 20 participants, although it is possible to use any number, from one to hundreds. After the participants have been selected, the initiator works with them to develop the focus of the conceptualization. Some examples of conceptualization focus for programme evaluation in are given in [40]: the nature of the policy, the desired outcomes, or the type of people to be included in the evaluation.

Generation of Statements

Once the focus becomes clear, a set of statements within the focus is generated using brainstorming or alternative methods, such as brainwriting, nominal group techniques, focus groups, and qualitative text analysis (see also [43, 24, 32]). Trochim suggests that

the number of statements should be kept to a manageable level. According to Trochim's model for the conceptualization process, it is possible also to generate the statements one by one (e.g., by selecting from a predefined list of statements).

Structuring of Statements

Once the statements are ready, they are printed onto cards and given to each participant. Then a technique called card sorting [36] is applied. Each participant is instructed to group the cards "in a way that makes sense to you". Trochim places several restrictions on this procedure: each statement can only be placed in one group (i.e., an item cannot be placed in two groups simultaneously); there must be at least two groups of statements; and, at least one group must have more than one statement. If the participants perceive that there are several different ways to group the cards, it is possible to have them select the most sensible arrangement, or to record several groupings for each participant. When the grouping task is completed, the results are combined across people by using the following three steps:

1. The groupings made by each participant are recorded in a binary symmetric similarity matrix (see Figure 6.1:).

2. The individual similarity matrices are combined into an aggregated similarity matrix. Each of the values in this matrix represents how many participants placed a given pair of statements together.

3. Each statement is rated according to the rating focus, using a five- or seven-item Likert.

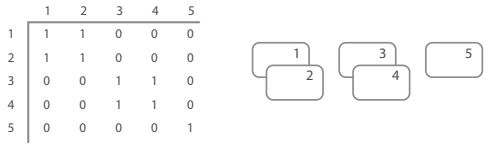

Figure 6.1: Example of a similarity matrix (left) for five statements grouped into three piles (right)

The aggregated similarity matrix provides information about how the participants grouped the statements; hence, it provides a representation of the relational structure of the conceptual domain. A high value implies that the statements are conceptually similar in some respect. A low value implies that they are conceptually more distinct. Both the generation and structuring of statements could involve individuals, groups, or the usage

of non-subjective criteria, such as relevant theories or predefined algorithms (e.g., cluster analysis).

Representation of Statements

In the representation stage, the grouping and rating input is represented pictorially by using two statistical analyses: First, a *two-dimensional multidimensional scaling* [23] takes the grouping data across all participants and develops a *point map* in which each statement becomes a point on the map. The more people that have grouped the same statements together, the closer the statements are to each other on the map.

Second, a *hierarchical cluster analysis* [12] takes the output of the multidimensional scaling (the *point map*) and partitions the map into groups of statements or ideas, forming clusters. If the statements describe activities of a programme, the clusters show how these can be grouped into logical groups of activities. If the statements represent specific outcomes of a policy/programme, the clusters might be viewed as outcome constructs or concepts. This second map is called "the conceptual domain of the outcomes".

Trochim suggests using Ward's hierarchical cluster analysis [12, 40], because it can generate several configurations of clusters, from which the participants can then decide which one makes the most sense. Note that the statistical methods described above are not the only ones available for generating a cluster map. However, they are the most suitable for *concept mapping*, and Trochim explains why in [40].

An additional result of the cluster analysis is the generation of a *cluster list* or a *named list of piles* [40] (into which the different statements are grouped), which is used in later stages of the *concept mapping* process. Finally, each of the points can be assigned a given value, which can be represented by a bar. For example, the value may correspond to the perceived importance or criticality of a statement. The height of the bar then indicates the aggregated importance of the issue for the complete set of participants who rated the statements.

Interpretation of Concept Maps

The facilitator should work with the participants to help them to develop their own labels and interpretations for the different maps. In this analysis session, the participants use the cluster list to choose names for the different clusters, by negotiating (similarly to naming factors in factor analysis). The maps are then used to adjust the naming of the clusters and adjust the clusters themselves.

Utilization of Concept Maps

The maps can be used as a visual framework to implement or evaluate a given programme. They can also be used as the basis for developing measures and displaying results. Each cluster can be seen as a construct. The individual statements can suggest specific operationalizations of that construct.

Outputs

Several artefacts are created from the concept mapping process, for example, the statement list or the list of brainstormed statements, the cluster list, and the different maps (i.e., point map, cluster map, point-rating map). Another artefact is a cluster-rating map, which consists of the cluster map with average cluster ratings overlaid. This specific type of map was not included in this work. Examples of all these artefacts can be found in [40].

3 Towards a Maintainability Map

We now describe how we used concept mapping to generate a concept map for analyzing maintainability. We analyzed four web applications written in Java that manage information about empirical studies conducted by Simula Research Laboratory. All four systems conform to the same requirements specification but have considerable dissimilarities. For instance, their size varies from 7208 LOC for the smallest system to 14549 LOC for the largest one. Four Norwegian consultancy companies developed the four systems independently. More details of the case study that was conducted on these development projects can be found in [1, 2, 6].

Using Trochim's concept mapping approach, we derived conceptual maps, using as a basis software measures and other indicators relevant to the systems under study. This process resulted in a tailored two-dimensional point map that was composed on the basis of the measures, which were clustered into constructs that describe maintainability.

3.1 Preparation

The participants were four software engineering researchers and one professional software engineer. All had good knowledge of software maintenance in general and of the actual systems of analysis. The professional software engineer had more than 25 years of experience with software development. The researchers all had professional experience of software development, as well as good knowledge of code measures and design attributes. The participants represented a convenience sample, in that all the researchers were at the time associated with the same research group (2nd, 3rd, 5th and 6th authors of this

paper), and the engineer (4th author) had already evaluated the maintainability of the systems. The initiator was the 1st author. The focus question was: "Which characteristics could be used to better understand the maintainability of the four systems?"

3.2 Generation of Statements

Our process for generating statements deviated slightly from Trochim's original approach. We were interested in exploring different software indicators and how useful for uncovering or predicting maintainability issues it would be to combine them in different ways. Instead of performing a brainstorming session, we preselected a set of design attributes as statements for our concept mapping. A design attribute is here widely defined as a code attribute, code smell, or design principle violation that is present in the design of a software system.

We did not perform a brainstorming session in this case because we wanted to include only code smells and design principle violations that were formally defined and detectable automatically. (The term code smells, which was coined by Kent Beck and Martin Fowler [13], is informally defined as bad or inconsistent parts of the design of object-oriented software.) We used Borland Together 2008 [8] for this purpose. In addition, we used the software measures suggested in [6]. These measures were the result of an analysis to find a minimal set of measures that could describe the dimensions of design that influence maintainability in these four systems [6].

Moreover, we included different software measures as part of the statements, so that we could use the knowledge derived from empirical studies on measures of software structural attributes [19, 46, 14]. Arisholm and Sjøberg [3] suggest that metrics may be more practical when used in combination than when interpreted individually.

However, from a practical point of view, analysis that relies only on software measures does not offer clear guidance to developers about how to improve maintainability. To overcome this limitation, we incorporated code smells and design principle violations (defined as structural symptoms) to complement the analysis of software measures.
Our motivation for integrating structural symptoms into the list of statements is that for each of these, redesign strategies (e.g., refactoring, use of design patterns) are available for improving the software [13]. There has been a growing interest in the topic of code smells for assessments of software maintainability. Van Emden and Moonen [45] provided the first formalization of code smells and described a tool that could detect them. Marinescu [28] further formalized the definition of code smells and extended the detection to a wider range of code smells and a number of design principle violations.

The resulting list of design attributes used as statements is presented in Table 6.1. It shows for each statement, its number, the name of the corresponding design attribute

and the type of design attribute, where 'CS' means Code Smells, 'DPV' means Design Principle Violations, and 'CM' means Code Measures.

Table 6.1: List of statements (design attributes)

No.	Design Attribute	Type	No.	Design Attribute	Type
1	Interface segregation principle (ISP) violation	DPV	18	Refused bequest	CS
2	Data class	CS	19	Subclasses have the same member	CS
3	God method	CS	20	Feature envy	CS
4	Tight class cohesion (TCC)	CM	21	Suspicious usage of switch statements	CS
5	Temporary variable is used for several purposes	CS	22	Import list construction	CS
6	Comments: Lines of comments in the code	CM	23	Field is used as a temporary variable	CS
7	Usage of implementation instead of interface	DPV	24	Number of children (NOC)	CM
8	Shotgun surgery	CS	25	Unused local variable or formal parameter	CS
9	Call from methods in an unrelated class (OMMEC)	CM	26	Duplicated code in constructors	CS
10	Depth of inheritance tree (DIT)	CM	27	Member is not used	CS
11	Data clump	CS	28	Call to methods in an unrelated class (OMMIC)	CM
12	Long message chain	CS	29	God class	CS
13	Number of lines of code (LOC)	CM	30	Declaration is hidden	CS
14	Single responsibility principle	DPV	31	Dead code	CS
15	Unused class	CS	32	Wide subsystem interface (lack of façade)	DPV
16	Duplicated code in conditional branches	CS	33	Number of methods in a class (WMC)	CM
17	Misplaced class	CS			

3.3 Structuring of Statements

The statements were structured in two steps. The first step consisted of a discussion and elicitation session. The second step consisted of grouping the statements. The main purpose of the discussion and elicitation session was to discuss the implications of the selected software design attributes on the systems, and how they might be interrelated or grouped together according to different perspectives. Given that the perspectives used for the grouping could be diverse, the participants were required to give reasons for grouping the statements in the way they did. For instance, some people might relate or group measures using base-rate, relationship, or causality viewpoints, whereas others may relate two attributes using a risk management viewpoint (e.g., "grouping the most risky ones

together").

All these perspectives were considered to be valid, given that our goal was to find the perspectives that were most relevant for the evaluation of the systems according to the type of system and type of maintenance tasks that were planned for them. The first step started with an explanation of the session. The list of design attributes was then presented to the participants. For each of the attributes, a discussion was held about: (a) what effects it might have on the system, (b) what the reason was behind their presence (their nature), (c) how they were related to other attributes from a certain point of view, and (d) whether any other perspectives could be applied.

The facilitator compiled the comments from the participants. After the session, the participants were assumed to have acquired enough contextual information, as well as a reasonable perception of how these different measures and smells could be grouped. For the second step, the participants were requested to group the set of design attributes by using a web tool [11].

3.4 Representation of Statements

The purpose of this session was to obtain group agreement on the aspects (which were represented in the names of the clusters) of maintainability that should be used in order to interpret the Code Smells, Design Principle Violations, and Code Measures. The outcome of the session should be a point map and a cluster map, composed of a set of design attributes that represents different aspects of the software design. Given that the participants made several groupings according to different perspectives, a second session was needed in order to synchronize the participants' interpretation of the perspectives and the names of the clusters. Normally, only one perspective is used for concept mapping, but we initially allowed several perspectives (or several groupings, each using a different perspective) to enable the one(s) that is most important for the assessment to be selected.

During the session, the cluster lists for the participants were presented and discussed, in order to group the different cluster names together according to a given perspective. For instance, one of the chosen perspectives of two of the participants was the "Severity" of the design attributes. One of the two used "Severe", "Moderate", "Dependent of Context" and "Low" as cluster names for grouping the statements. The other used "Serious", "Suspicious", "Unimportant" and "Need a balance" as cluster names.

In cases in which the perspective was not sufficiently clear, the sets of individual measures that belonged to the clusters were compared in order to see if they were similar, in which case it was assumed that the clusters represented similar concepts. Table 6.2 lists the perspectives that were generated after the discussion. We focused on "Design/Structural issues" because that allowed us to compare the results of our approach with other

available quality models [21] that share the same perspective. We aggregated the groupings that used the "Design/Structural issues" perspective using a tool [11] that implements both multidimensional scaling and hierarchical cluster analysis. The resulting point map is shown in Figure 6.2. The outcome from the hierarchical cluster analysis (a cluster map) is shown in Figure 6.3.

Table 6.2: List of perspectives drawn from discussion

Perspective	**Description**
Severity	The level of risk they might represent (e.g., from "Severe" to "Low")
Predictability	The level of uncertainty they represent (also following a risk-assessment perspective)
Properties of developers	Human factors that indicate the type of developers who implemented the system
Design/Structural	Software design concepts
Abstraction level	Level of responsibility for the required refactoring or redesign (e.g., "Programmer's responsibility", "Designer's responsibility" and "Architect's responsibility")
Cause indicators	Representing potential events/factors that lead the system to display these attributes (e.g., "Bad coding practices", "Design unused or used for informal testing")

Figure 6.2: Point map derived by our concept mapping

As part of the participants' grouping of the statements in piles (see Section 3.3), they were asked to come up with a representative name for each of the piles. The cluster analysis algorithm selects a name for each of the clusters on the basis of the names proposed initially by the participants to designate each of the piles. The naming is adjusted accordingly during the negotiation stage, which is described in Section 2.3.

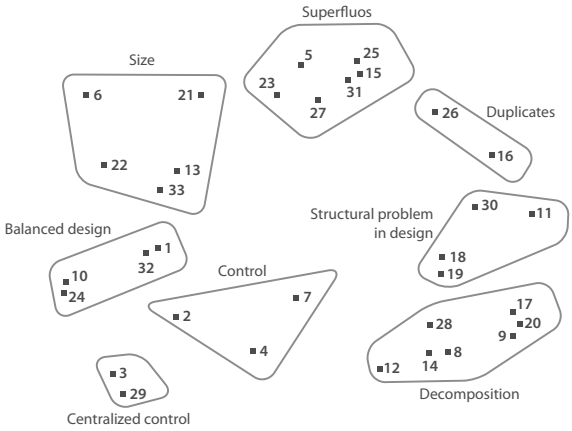

Figure 6.3: Cluster map generated by Concept System

3.5 Interpretation of Concept Maps

The remainder of the session aimed at adjusting the cluster map. Clusters representing similar concepts and measures were identified (see Columns 1 and 2 in Table 6.3), and names for common concepts were agreed upon. The map that resulted from the discussion is shown in Figure 6.4.

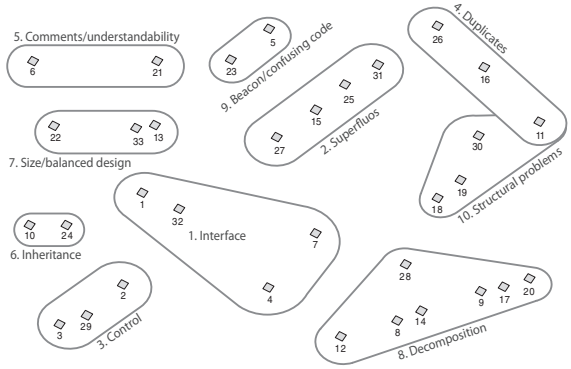

Figure 6.4: Final cluster map after discussion

In some cases, it was not possible to find an intersection of all the conceptually equivalent piles. For instance, in cluster 7 “Decomposition”, there are common attributes between clusters of the participants P1, P3, and P4, but the intersection would be empty if the cluster of participant P2 were considered. In those cases, the nearness of the attributes in the point map or in the previously generated cluster map was used as a basis for deciding which attributes to include in the final cluster. In cases where there was no final agreement

on the cluster to which an attribute should belong, the issue was left open, as in the case of cluster 4 “Duplicates” and cluster 10 “Structural problems”, which share Attribute 11 (Figure 6.4).

Table 6.3: Cluster names chosen for distinguishing the conceptually equivalent clusters, the statements per participant per cluster, and the final statements of the cluster after the negotiation stage

Cluster name	**Participant**	**Statements of the equivalent piles**	**Statements**
Interface/Balanced design	P1	1,7,30,32	1,4,7,32
	P2	1,7,22,27,30,32	
	P3	1,2,4,7,22,26,32	
Superfluous	P1	15,25,27,31	15,25,27,31
	P4	15,16,21,22,25,26,27	
	P5	15,18,25,27,31	
Control	P1	3,12,23,29	2,3,29
	P4	3,6,29	
	P2	2,3,4,12,14,29	
	P3	3,10,24,29	
Duplicates	P1	2,5,11,16,26	11,16,26
Comments/Understandability	P1	6,21,22	6,21
Inheritance/Balanced design	P1	4,10,13,24,28,33	10,24
	P4	1,2,10,13,18,19,24,32,33	
Size/Balanced design	P1	4,10,13,24,28,33	13,22,33
	P4	1,2,10,13,18,19,24,32,33	
Decomposition	P1	8,9,14,17,18,19,20	8,9,12,14,17,20,28
	P4	4,7,8,9,11,12,14,17,20,28	
	P2	10,18,19,21,24	
	P3	1,2,4,7,22,26,32	
Beacons/Confusing code	P4	5,23,30	5, 23
	P3	6,9,13,33	
	P2	5,6,13,15,21,23,25,27,31,33	
Structural problems	P5	9,10,17,19,20,21,22,26	11,18,19,30
	P3	8,9,11,12,14,16,17,18,19,20,28,30	

3.6 Utilization of Concept Maps

We retrieved the values for the measures, using Borland Together 2008 [8]. To compare measures on different scales, the measures were standardized across the values from each of the systems (see Figure 6.5). Due to the fact that the measures within the clusters do not have a common measurement unit (e.g., clusters can contain smells, measures, and

design violations), using a cluster-rating map would not be adequate for our analysis. A cluster-rating map does not allow for the interpretation of individual indicators. In addition, displaying a mean value of measures with different units would make it difficult to interpret the mean values correctly.

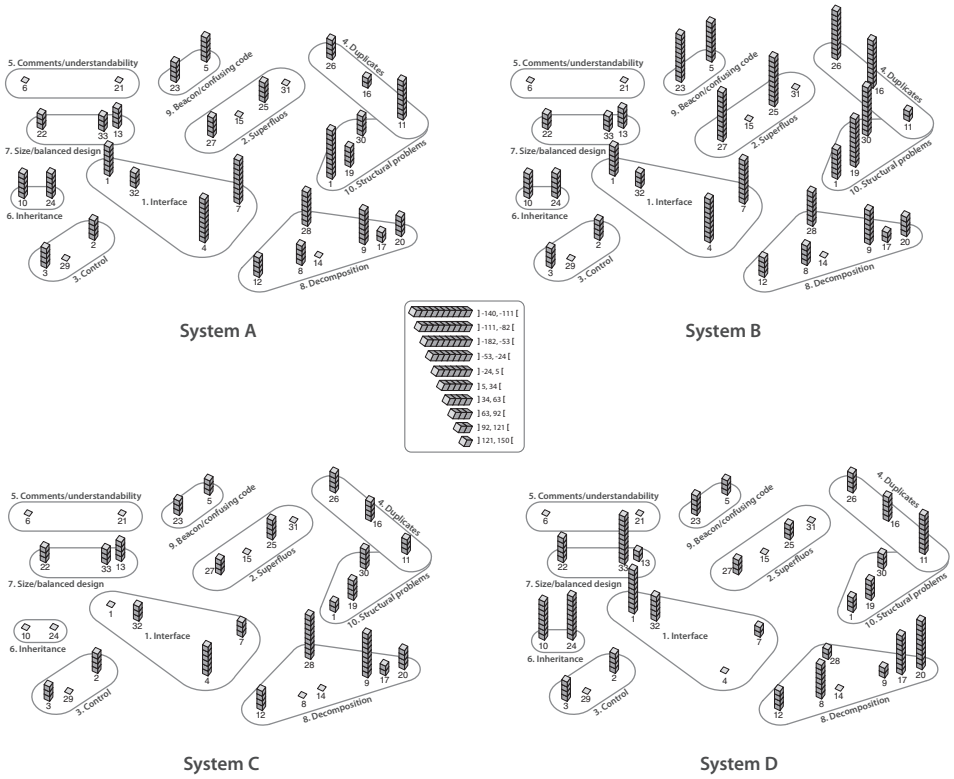

Figure 6.5: Cluster map with measurements from Systems A, B, C and D

Instead of using a cluster-rating map, we used a combination between a cluster-map and point-rating map, in order to compare the individual measurement values across the different clusters. We represent the relative difference between the values in the form of a 10-scale value (with the smallest value found being -1.4 (one block) and highest value being 1.5 (nine blocks)) and assign five blocks to the closest values to the mean[1]. The motivation for using a 10-scale bar was that it gives us the right level of granularity for visualizing the differences between the systems. We used the four maps to analyze the differences between the four systems. Concept maps may also be used to generate hypothetical patterns in

[1]From the set of standardized values of the measures across the 4 systems, -1.4 was approximately the smallest value and 1.5 was the highest value.

pattern matching [47]. Pattern matching requires drawing a theoretical pattern of expected outcomes (comparable to a hypothesis) on the basis of a pattern of characteristics of a given object or phenomenon (similar to control variables). While experimental studies typically involve univariate analysis, pattern matching follows the multivariate analysis perspective, which is very suited to case studies. See our plans with respect to pattern matching in Section 5.

3.7 Outcome and General Remarks

The findings from the mapping process and subsequent analysis will be presented in the following subsections.

Structuring of Expert Knowledge

To exemplify how expert knowledge in this case was structured and operationalized through concept mapping, we compared the results from the expert evaluation reported in [1] and the resulting concept maps. One example is the comments from the expert about System B using a comprehensive proprietary library. This may be viewed as relating to cluster 6.2 Size/Balanced design (measure 22: Import list construction). This cluster also indicates considerable differences in their size (A has 7937 LOC vs. B with 14549 LOC). Cluster 3 Control has a considerable difference between the systems for Measure 2: Data Class. In addition, in the Control cluster, system B has many more God Methods than System A, which could be related to the factors to which the experts referred in Choice of classes from [1]:

A	*"Contains primary objects that are implemented with classes that contain both data and logic."*
B	*"Has primary objects, which are implemented as containers, and they also remark that there are additional, unnecessary containers."*

Another example is the description by the experts regarding *Inheritance*, where they stated:

A	*"Mostly successful use of inheritance, but in some cases the base class does not contain all the functionality that is expected in a base class."*
B	*"Too extensive use of inheritance. Confusions regarding whether functionality should be in the base class or the subclass."*

This difference between the systems can be seen in Cluster: *Inheritance/Balanced design* where the values for Measures 10 (*Depth of Inheritance Tree*) and 24 (*Number of Children*) are considerably higher in B. We found that the use of objective observations (in this

case software measures) helps to reduce misinterpretations, which are often the result of evaluations described informally in unstructured text.

Interpretation from Different Perspectives

The perspective chosen by most of the researchers was different from the one chosen by the professional software engineer. Where the prevalent perspective of the researchers was similar to the taxonomy by Mäntylä [26], the dimensions used by the software engineer related more to the predictability levels of each of the attributes, as well as the potential severity they represent (i.e., they adopted what may be termed a risk analysis perspective)[2].

From the discussion, it became clear that these two perspectives were strongly intertwined. That being so, we repeated the process, this time choosing a combination between Severity and Predictability perspectives. The result is shown in Figure 6.6, where the clusters have more or less two major segments, one side with the unpredictable/serious indicators and the other side with the measures deemed not so serious.
An interesting aspect is that the cluster Maintenance Risk Depends on Context is composed mainly of software measures and code smells that are related to size and control (e.g., God Class, God Method), which also reflects the need for additional detail in order to categorize these indicators as harmful or not. A God Class may appear to be the only option for architectures that require centralized control. However, of course, this is not clear from just looking at the characteristics of the system. Therefore, an additional map that describes the context of the project (which could state the nature of the maintenance tasks, the developers' skills, and the requirements of the system) would be needed to evaluate these indicators.

Although systems A and B have relatively similar values in the cluster Maintenance Risk Depends on Context (which might indicate that either systems could 'behave nicely' depending of the setting), the high values from system B on the clusters Serious and Potentially time consuming should raise a flag to the evaluators if they want to consider system B for a project with very strict deadlines. Due to limitations of space, we are unable to present details regarding the results from this analysis.

4 Benefits of Concept Mapping

We discuss four subareas of software assessment in which concept mapping may improve on current approaches, state some of the challenges that it faces, and offer recommendations for using this method in practice.

[2]The severity perspective was also chosen by two of the researchers.

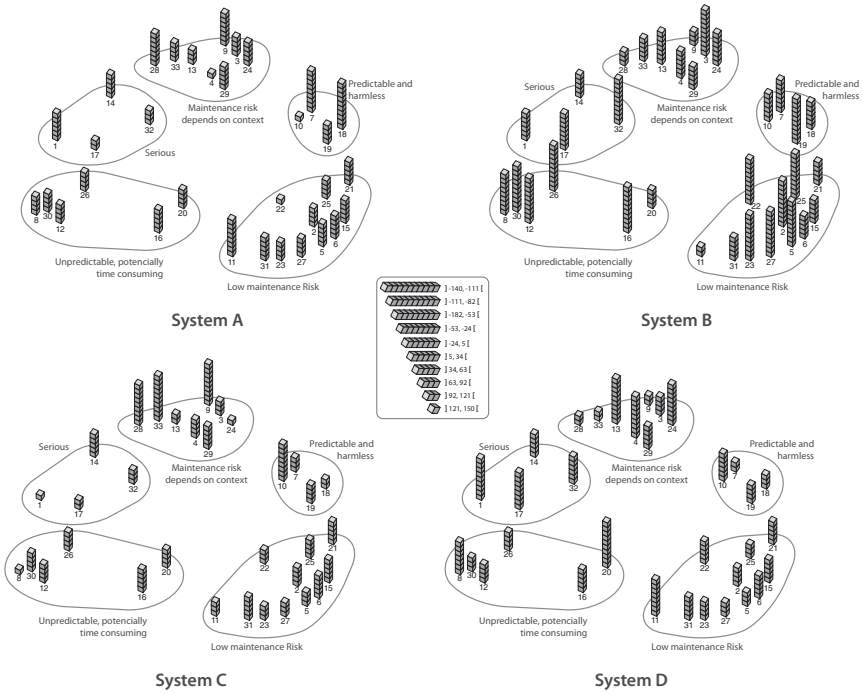

Figure 6.6: Cluster map with measurements from Systems A, B, C and D using the *severity/uncertainty* perspectives

4.1 Transparent Quality Models

Numerous models and frameworks have been proposed for evaluating software quality in industrial settings. For instance, in the Goal Question Metric [5] paradigm, the measurement models are derived by linking measurement goals to operational questions, which can be answered by measuring aspects of products, processes, or resources.

Other models follow the hierarchical approach [17, 4, 34] and breakdown external quality attributes into internal attributes and from there to low-level measures. Examples of other approaches to quality models that follow the Factor Criteria Metric paradigm [30] are the complexity model for object-oriented systems by Tegarden et al. [39] and the Maintainability Index (MI) by Oman and Hagemeister [34], which is typically used for evaluating maintainability at system level.

However, as Marinescu points out [27], many of these approaches have an implicit mapping between observable measures and the abstract attributes; hence, the criteria that are used for the mapping are not made explicit. In many cases, neither the criteria nor the process followed for deriving these criteria are stated. Part of the focus with these models has been on establishing empirical validation of the mapping criteria, but again, many studies have limited generalizability due to the lack of contextual information and the inherent complexity of the projects.

Concept mapping helps to provide explicit descriptions of these mapping criteria and the process by which they are derived. The mapping criteria can later be adapted and improved, using empirical evidence and expert knowledge as a basis. Another advantage with concept mapping is that it provides a pictorial representation, where the measures within the attributes are not concealed. This allows us to observe the measures in relation to all other measures, and to see how each of them fit into the overall picture. In our case, we found that multivariate representations support better data comparison across systems and thus facilitate data interpretation. For instance, if we compare Kiviat charts to concept maps, we find that the latter are more scalable in terms of number of displayable measures, thereby providing better use of the 2D space for representing the relations between the different measures.

4.2 Developing Tailored Quality Models

The ISO/IEC 9126 standard [16] outlines software quality attributes and decomposes them into subattributes in a hierarchical way. While this can be a good starting point, the operationalizations of the attributes are specified only partially. Concept mapping could guide the operationalization by starting from low-level properties (i.e., the statements) and inferring their meaning according to a given goal or purpose. For example, if we

intended to undertake adaptive maintenance, we could group the measures according to such aspects as interfaces, separation of concerns, and level of definition of architectural layers, and link them back to the higher-level attribute of adaptability. Another example of how to use concept mapping is to start from the quality attributes of ISO (five in addition to maintainability) and use them as focus topics. Groups of experts could then generate statements that operationalize the five high-level attributes, as we have done for maintainability.

In the same way as ISO, ontologies such as the one by Kitchenham et al. [22] identify and describe a series of factors that were thought to affect maintenance and empirical studies of maintenance. Kitchenham's ontology outlines a wide range of concepts, and it is still a challenge to identify which are the relevant ones for a given purpose. More specialized ontologies have been developed, for example, for describing software maintenance and evolution [20] and for performing software design analysis [26]. This illustrates the need to adapt quality models into different domains of concern [21]. Concept mapping can help the construction of domain- and context-specific ontologies that is based on the knowledge and experience of experts. In a given area, domain experts may have other indicators that are more comprehensive than the ones in the standard quality models (See Li and Smidts [31]). Concept mapping could be useful for identifying the indicators that the experts use. Given that concept mapping requires that a focus be chosen for the generation and structuring of the statements, it will guide the selection of relevant aspects in order to derive a framework for assessment, ontology, or quality model.

4.3 Representing Contextual Information

Throughout the paper, we have emphasized the importance of contextual information for assessment purposes. Mayrand and Coallier [29] exemplify how to incorporate contextual information by describing a process-oriented procurement project that combines capability assessment with static analysis. The authors point out that the lack of contextual description makes it difficult to interpret the metrics. For instance, from a metric point of view, a decision tree may seem complex, but from the programmer's point of view, the logic is well organized and easy to understand and validate. Briand and Wüst [9], and Bengtsson and Bosch [7] have incorporated contextual information by integrating *change scenarios* with software measures. Correspondingly, Van Deursen and Kuipers [44] propose a method of risk assessment that is based on software product and process analysis.

Concept mapping can be used to generate representations, not only of systems, but also of different projects. The latter category of maps might be equivalent to Trochim's programme-characteristic pattern [40]. Having a visual representation of contextual data will enable the decision-makers to consider the different factors involved in the project

and consequently interpret the relative importance of the different characteristics of the system. Carr and Wagner [10] report that experts consider contextual aspects, such as the cost, urgency, and difficulty of a maintenance task, while making decisions concerning how a given maintenance goal or need is, respectively, to be achieved or met.

4.4 Incorporating Expert Knowledge

We have proposed the use of expert judgment in conjunction with concept mapping. Expert judgment in software engineering has been used for management and decision-making purposes [18, 37]. Moreover, Carr and Wagner [10] reported that experts use case-based reasoning and heuristics to prioritize maintenance projects. The heuristics that experts use are the result of knowledge gained from experience of past projects, in which different problems were solved with different solutions. As we noted in Section 3.7, concept mapping can help to structure and accumulate expert knowledge.

Heitlager et al. [15] stresses the need for cost-effective measurement frameworks that support root-cause analysis to identify specific problems and connect them to specific solutions. One reason why Design Patterns and Code Smells/Refactorings have gain popularity, is that they map a solution to a specific problem [13]. At a higher level of abstraction, concept mapping can be used to develop pattern-based evaluations, following the same line of thought as design patterns. By using the opinions of experts as input, we can use concept mapping to generate representations of both project contexts and the objects under evaluation. This could guide the process of defining the most desirable characteristics in a system for particular contexts.

5 Challenges and Limitations

The main limitation of concept mapping is that any method that uses experts relies on the availability of sufficiently qualified experts. It might be thought that diverging opinions or lack of knowledge among the participants would be a problem. However, in a practical setting, the participants should be system stakeholders. That being so, strong divergence of opinion or a lack of the qualifications necessary to make good judgments about the system's maintainability might be a good sign that there are problems inherent in the project; hence, divergence of opinion and lack of knowledge do not necessarily constitute a limitation of concept mapping. Although concept mapping is not particularly time-consuming, the use of this method does generate some overhead, which needs to be weighted alongside the ISO standard or GQM. However, note that even when using the well-established methodologies, interpreting metrics might take considerable time. We are aware that in some situations, a simple model might be better suited to solving the

problem than concept mapping. However, when we introduce concept mapping into the software engineering discipline, it is under the assumption that there are many similarities between maintainability assessments and programme evaluation, and in the programme evaluation domain it is evident that a simple model does not suffice for evaluation purposes.

A limitation in the concept mapping process that we report herein was the low number of participants (5) compared with the number prescribed by Trochim. In addition, we are aware that a prescribed set of statements (the design attributes) could result in a loss of important contextual information for the evaluation. In order to address this limitation, an additional focus could have been to identify the contextual aspects that were relevant to the maintenance project (e.g., the type of maintenance tasks) so that these could have been considered explicitly during the discussion/generation/grouping of the statements.

6 Conclusions and Future Work

The software engineering industry has a strong need for methods to assess software maintainability that go beyond what evaluation purely based on Code Measures can provide. We find that concept mapping constitutes a strong addition to existing frameworks for evaluating quality, such as ISO/IEC 9126 and GQM, and tools for static measurement of software code. Our overall experience is that concept mapping provides a systematic, structured, and repeatable method for developing constructs and measures of the phenomenon of interest. It is useful for defining constructs and measures of other aspects of software engineering, in addition to maintainability.

Some areas for future work pertain to deriving an efficient and reliable method for providing human assessment in software maintainability using concept mapping. The model needs to be efficient if it is to gain acceptance in the industry, because the time spent on using this approach will be weighed against the gain in efficiency and risk assessment in the overall maintenance project. Reliability and accuracy can be improved by ensuring a consistent interpretation of the grouping activities and subsequent negotiating process, thereby reducing the chance for misinterpretation and misunderstanding. Incorporating repeated expert assessments of various maintenance tasks might provide a valuable general knowledge base that can be used in combination with specific contextual input in the maintenance project to be evaluated. This would make it possible to extend tools that rely on evaluation of Code Measures to include context-specific assessment of software maintainability.

Our most immediate plan is to use pattern matching to analyze the data from a multiple-case study in which several maintenance tasks were performed on each of four

systems (two of which are described in this article). In this case, the descriptive maps of the systems are the control variables. Since we need to generate a hypothetical pattern, another concept mapping session was conducted to generate a pattern for the outcomes of maintenance projects (called process outcome pattern by Trochim [41]). This pattern comprises different aspects that could describe the outcomes from a given project, in our case representing the outcomes from the multiple-case study. We drew four different outcome patterns, using our assumptions for the results from each of the four systems maintenance projects. We expect that this type of data analysis will prove useful for determining the usefulness of different measures for predicting outcomes from maintenance projects.

7 References

[1] Bente C. D. Anda. "Assessing Software System Maintainability using Structural Measures and Expert Assessments." In: *Int'l Conf. Softw. Maint.* 2007, pp. 204–213.

[2] Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System." In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 407–429.

[3] Erik Arisholm and Dag I. K. Sjøberg. "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software." In: *IEEE Transactions on Software Engineering* 30.8 (2004), pp. 521–534.

[4] Jagdish Bansiya and Carl G. Davis. "A Hierarchical Model for Object-Oriented Design Quality Assessment." In: *IEEE Transactions on Software Engineering* 28.1 (2002), pp. 4–17.

[5] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. "Goal Question Metrics Paradigm." In: *Encyclopedia of Software Engineering*. Vol. 1. John Wiley and Sons, Inc., 1994, pp. 528–532.

[6] Hans Benestad, Bente Anda, and Erik Arisholm. "Assessing Software Product Maintainability Based on Class-Level Structural Measures." In: *Product-Focused Softw. Process Improvement*. 2006, pp. 94–111.

[7] Per Olof Bengtsson and Jan Bosch. "An experiment on creating scenario profiles for software change." In: *Annals of Software Engineering* 9.1–4 (2000), pp. 59–78.

[8] Borland. *Borland Together [online] Available at: http://www.borland.com/us/products/together [Accessed 10 May 2012]*. 2012.

[9] Lionel C. Briand and Jürgen Wüst. "Integrating scenario-based and measurement-based software product assessment." In: *Journal of Systems and Software* 59.1 (2001), pp. 3–22.

[10] Mahil Carr and Christian Wagner. "A Study of Reasoning Processes in Software Maintenance Management." In: *Journal of Information Technology and Management* 3.1–2 (2002), pp. 181–203.

[11] Concept-Systems. *Concept Systems [online] Available at: http://www.conceptsystems.com [Accessed 10 March 2009].* 2009.

[12] Brian Everitt. *Cluster analysis*. 2nd Edition. New York: Halsted Press, A Division of John Wiley and Sons, 1980.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] Marcela Genero, Daniel L. Moody, and Mario Piattini. "Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation." In: *Journal of Software Maintenance* 17.3 (2005), pp. 225–246.

[15] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability." In: *Int'l Conf. Quality of Inf. and Comm. Techn.* 2007, pp. 30–39.

[16] ISO/IEC. *International Standard ISO/IEC 9126*. Tech. rep. Geneva: International Organization for Standardization, 1991.

[17] ISO/IEC. *Software Engineering - Product quality - Part 1: Quality model*. Tech. rep. 2001.

[18] Magne Jørgensen. "A review of studies on expert estimation of software development effort." In: *Journal of Systems and Software* 70.1–2 (2004), pp. 37–60.

[19] Dennis Kafura and Geereddy. R. Reddy. "The Use of Software Complexity Metrics in Software Maintenance." In: *IEEE Transactions on Software Engineering* 13.3 (1987), pp. 335–343.

[20] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. "Mining Software Repositories with iSPAROL and a Software Evolution Ontology." In: *Int'l Workshop on Mining Software Repositories*. 2007, p. 10.

[21] Barbara Kitchenham et al. "The SQUID approach to defining a quality model." In: *Software Quality Control* 6.3 (1997), pp. 211–233.

[22] Barbara A. Kitchenham et al. "Towards an ontology of software maintenance." In: *Journal of Software Maintenance* 11.6 (1999), pp. 365–389.

[23] Joseph B. Kruskal and Myron Wish. *Multidimensional Scaling*. Sage University Paper series on Quantitative Applications in the Social Sciences. Newbury Park, CA: Sage Publications, 1978.

[24] Mary C. Lacity and Marius A. Janson. "Understanding qualitative data: a framework of text analysis methods." In: *Journal of Management Information Systems* 11.2 (1994), pp. 137–155.

[25] Stanley Letovsky. "Cognitive processes in program comprehension." In: *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Ablex Publishing Corp., 1986, pp. 58–79.

[26] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. "A taxonomy and an initial empirical study of bad smells in code." In: *Int'l Conf. Softw. Maint.* 2003, pp. 381–384.

[27] Radu Marinescu. "Measurement and Quality in Object Oriented Design." Doctoral Thesis. "Politehnica" University of Timisoara, 2002.

[28] Radu Marinescu. "Measurement and quality in object-oriented design." In: *Int'l Conf. Softw. Maint.* 2005, pp. 701–704.

[29] Jean Mayrand and Francois Coallier. "System acquisition based on software product assessment." In: *Int'l Conf. Softw. Eng.* 1996, pp. 210–219.

[30] Jim A. McCall, Paul G. Richards, and Gene F. Walters. *Factors in Software Quality*. Vol. I. NTIS, 1977.

[31] Li Ming and C. S. Smidts. "A ranking of software engineering measures based on expert opinion." In: *IEEE Transactions on Software Engineering* 29 (2003), pp. 811–824.

[32] David L. Morgan. *Focus Groups as Qualitative Research*. Newbury Park, CA: Sage Publications, 1988.

[33] Michael O'Brien. *Software comprehension: A review and research direction*. Technical Report no. UL-CSIS-03-3. University of Limerick, 2003.

[34] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability." In: *Int'l Conf. Softw. Maint.* 1992, pp. 337–344.

[35] Markus Pizka and Florian Deissenboeck. "How to effectively define and measure maintainability." In: *Softw. Measurement European Forum*. 2007.

[36] Seymour Rosenberg and Moonja Park Kim. "The Method of Sorting as a Data-Gathering Procedure in Multivariate Research." In: *Multivariate Behavioral Research* 10.4 (1975), pp. 489–502.

[37] Tony Rosqvist, Mika Koskela, and Hannu Harju. “Software Quality Evaluation Based on Expert Judgement.” In: *Softw. Quality Control* 11.1 (2003), pp. 39–55.

[38] Peter H. Rossi, Mark W. Lipsey, and Howard E. Freeman. *Evaluation: A Systematic Approach*. Thousand Oaks, CA: SAGE, 2004.

[39] David P. Tegarden, Steven D. Sheetz, and David E. Monarchi. “A software complexity model of object-oriented systems.” In: *Journal of Decision Support Systems* 13.3–4 (1995), pp. 241–262.

[40] William M. K. Trochim. “An introduction to concept mapping for planning and evaluation.” In: *Evaluation and program planning* 12.1 (1989), pp. 1–16.

[41] William M. K. Trochim. “Outcome pattern matching and program theory.” In: *Evaluation and Program Planning* 12.4 (1989), pp. 355–366.

[42] William M. K. Trochim and Rhoda Linton. “Conceptualization for planning and evaluation.” In: *Evaluation and Program Planning* 9.4 (1986), pp. 289–308.

[43] Andrew H. Van de Ven and Andre L. Delbecq. “The Effectiveness of Nominal, Delphi, and Interacting Group Decision Making Processes.” In: *The Academy of Management Journal* 17.4 (1974), pp. 605–621.

[44] Arie Van Deursen and Tobias Kuipers. “Source-Based Software Risk Assessment.” In: *Int’l Conf. Softw. Maint.* 2003, p. 385.

[45] Eva Van Emden and Leon Moonen. “Java quality assurance by detecting code smells.” In: *Working Conf. Reverse Eng.* 2001, pp. 97–106.

[46] Li Wei and Henry Sallie. “Object-oriented metrics that predict maintainability.” In: *Journal of Systems and Software* 23.2 (1993), pp. 111–122.

[47] Robert Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.