Université de Montréal

Impacts and Detection of Design Smells

par
Abdou Maiga

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Août, 2012

© Maiga, 2012

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Impacts and Detection of Design Smells

présentée par :

Abdou Maiga

a été évaluée par un jury composé des personnes suivantes :

Président-rapporteur	:	Philippe Langlais
Directrice de recherche	:	Esma Aïmeur
Codirecteur	:	Yann-Gaël Guéhéneuc
Membre du jury	:	Bruno Dufour
Examinateur externe	:	Ahmed E. Hassan
Représentant du doyen de la FAS	:	Philippe Langlais

Résumé

Les changements sont faits de façon continue dans le code source des logiciels pour prendre en compte les besoins des clients et corriger les fautes. Les changements continus peuvent conduire aux défauts de code et de conception. Les défauts de conception sont des mauvaises solutions à des problèmes récurrents de conception ou d'implémentation, généralement dans le développement orienté objet. Au cours des activités de compréhension et de changement et en raison du temps d'accès au marché, du manque de compréhension, et de leur expérience, les développeurs ne peuvent pas toujours suivre les normes de conception et les techniques de codage comme les patrons de conception. Par conséquent, ils introduisent des défauts de conception dans leurs systèmes. Dans la littérature, plusieurs auteurs ont fait valoir que les défauts de conception rendent les systèmes orientés objet plus difficile à comprendre, plus sujets aux fautes, et plus difficiles à changer que les systèmes sans les défauts de conception. Pourtant, seulement quelques-uns de ces auteurs ont fait une étude empirique sur l'impact des défauts de conception sur la compréhension et aucun d'entre eux n'a étudié l'impact des défauts de conception sur l'effort des développeurs pour corriger les fautes.

Dans cette thèse, nous proposons trois principales contributions. La première contribution est une étude empirique pour apporter des preuves de l'impact des défauts de conception sur la compréhension et le changement. Nous concevons et effectuons deux expériences avec 59 sujets, afin d'évaluer l'impact de la composition de deux occurrences de Blob ou deux occurrences de spaghetti code sur la performance des développeurs effectuant des tâches de compréhension et de changement. Nous mesurons la performance des développeurs en utilisant: (1) l'indice de charge de travail de la NASA pour leurs efforts, (2) le temps qu'ils ont passé dans l'accomplissement de leurs tâches, et (3) les pourcentages de bonnes réponses. Les résultats des deux expériences ont montré que deux occurrences de Blob ou de spaghetti code sont un obstacle significatif pour la performance des développeurs lors de tâches de compréhension et de changement. Les résultats obtenus justifient les recherches antérieures sur la spécification et la détection des défauts de conception. Les équipes de développement de logiciels doivent mettre en garde les développeurs contre le nombre élevé d'occurrences de défauts de conception et recommander des refactorisa-

tions à chaque étape du processus de développement pour supprimer ces défauts de conception quand c'est possible.

Dans la deuxième contribution, nous étudions la relation entre les défauts de conception et les fautes. Nous étudions l'impact de la présence des défauts de conception sur l'effort nécessaire pour corriger les fautes. Nous mesurons l'effort pour corriger les fautes à l'aide de trois indicateurs: (1) la durée de la période de correction, (2) le nombre de champs et méthodes touchés par la correction des fautes et (3) l'entropie des corrections de fautes dans le code-source. Nous menons une étude empirique avec 12 défauts de conception détectés dans 54 versions de quatre systèmes: ArgoUML, Eclipse, Mylyn, et Rhino. Nos résultats ont montré que la durée de la période de correction est plus longue pour les fautes impliquant des classes avec des défauts de conception. En outre, la correction des fautes dans les classes avec des défauts de conception fait changer plus de fichiers, plus les champs et des méthodes. Nous avons également observé que, après la correction d'une faute, le nombre d'occurrences de défauts de conception dans les classes impliquées dans la correction de la faute diminue. Comprendre l'impact des défauts de conception sur l'effort des développeurs pour corriger les fautes est important afin d'aider les équipes de développement pour mieux évaluer et prévoir l'impact de leurs décisions de conception et donc canaliser leurs efforts pour améliorer la qualité de leurs systèmes. Les équipes de développement doivent contrôler et supprimer les défauts de conception de leurs systèmes car ils sont susceptibles d'augmenter les efforts de changement.

La troisième contribution concerne la détection des défauts de conception. Pendant les activités de maintenance, il est important de disposer d'un outil capable de détecter les défauts de conception de façon incrémentale et itérative. Ce processus de détection incrémentale et itérative pourrait réduire les coûts, les efforts et les ressources en permettant aux praticiens d'identifier et de prendre en compte les occurrences de défauts de conception comme ils les trouvent lors de la compréhension et des changements. Les chercheurs ont proposé des approches pour détecter les occurrences de défauts de conception, mais ces approches ont actuellement quatre limites: (1) elles nécessitent une connaissance approfondie des défauts de conception, (2) elles ont une précision et un rappel limités, (3) elles ne sont pas itératives et incrémentales et (4) elles ne peuvent pas être appliquées sur des sous-ensembles de systèmes. Pour surmonter ces limitations, nous introduisons SMURF, une nouvelle approche pour détecter les défauts de conception, basé sur une technique d'apprentissage automatique — machines à vecteur de support — et prenant en compte les retours des praticiens. Grâce à une étude empirique portant sur trois systèmes et quatre défauts de conception, nous avons montré que la précision et le rappel de SMURF sont supérieurs à ceux de DETEX et BDTEX lors de la détection des occurrences de défauts de conception. Nous avons également montré que SMURF peut être appliqué à la fois dans les configurations intra-système

et inter-système. Enfin, nous avons montré que la précision et le rappel de SMURF sont améliorés quand on prend en compte les retours des praticiens.

Mots-clés : défaut de conception, anti-patrons, défauts de code, mauvaises odeurs, détection, restructurations, refactorisations, compréhension de programme, maintenance de programme, correction de faute, génie logiciel empirique.

Abstract

Changes are continuously made in the source code to take into account the needs of the customers and fix the faults. Continuous change can lead to *antipatterns* and *code smells*, collectively called “design smells” to occur in the source code. Design smells are poor solutions to recurring design or implementation problems, typically in object-oriented development. During comprehension and changes activities and due to the time-to-market, lack of understanding, and the developers’ experience, developers cannot always follow standard designing and coding techniques, *i.e.*, design patterns. Consequently, they introduce design smells in their systems. In the literature, several authors claimed that design smells make object-oriented software systems more difficult to understand, more fault-prone, and harder to change than systems without such design smells. Yet, few of these authors empirically investigate the impact of design smells on software understandability and none of them authors studied the impact of design smells on developers’ effort.

In this thesis, we propose three principal contributions. The *first contribution* is an empirical study to bring evidence of the impact of design smells on comprehension and change. We design and conduct two experiments with 59 subjects, to assess the impact of the composition of two Blob or two Spaghetti Code on the performance of developers performing comprehension and change tasks. We measure developers’ performance using: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. The results of the two experiments showed that two occurrences of Blob or Spaghetti Code design smells impedes significantly developers performance during comprehension and change tasks. The obtained results justify a posteriori previous researches on the specification and detection of design smells. Software development teams should warn developers against high number of occurrences of design smells and recommend refactorings at each step of the development to remove them when possible.

In the *second contribution*, we investigate the relation between design smells and faults in classes from the point of view of developers who must fix faults. We study the impact of the presence of design smells on the effort required to fix faults, which we measure using three metrics: (1) the duration of the fixing period; (2) the number of fields and methods impacted by fault-fixes;

and, (3) the entropy of the fault-fixes in the source code. We conduct an empirical study with 12 design smells detected in 54 releases of four systems: ArgoUML, Eclipse, Mylyn, and Rhino. Our results showed that the duration of the fixing period is longer for faults involving classes with design smells. Also, fixing faults in classes with design smells impacts more files, more fields, and more methods. We also observed that after a fault is fixed, the number of occurrences of design smells in the classes involved in the fault decreases. Understanding the impact of design smells on development effort is important to help development teams better assess and forecast the impact of their design decisions and therefore lead their effort to improve the quality of their software systems. Development teams should monitor and remove design smells from their software systems because they are likely to increase the change efforts.

The *third contribution* concerns design smells detection. During maintenance and evolution tasks, it is important to have a tool able to detect design smells incrementally and iteratively. This incremental and iterative detection process could reduce costs, effort, and resources by allowing practitioners to identify and take into account occurrences of design smells as they find them during comprehension and change. Researchers have proposed approaches to detect occurrences of design smells but these approaches have currently four limitations: (1) they require extensive knowledge of design smells; (2) they have limited precision and recall; (3) they are not incremental; and (4) they cannot be applied on subsets of systems. To overcome these limitations, we introduce SMURF, a novel approach to detect design smells, based on a machine learning technique—support vector machines—and taking into account practitioners' feedback. Through an empirical study involving three systems and four design smells, we showed that the accuracy of SMURF is greater than that of DETEX and BDTEX when detecting design smells occurrences. We also showed that SMURF can be applied in both intra-system and inter-system configurations. Finally, we reported that SMURF accuracy improves when using practitioners' feedback.

Keywords: design smells, antipatterns, code smells, bad smells, detection, restructuring, refactorings, program comprehension, program maintenance, fault fix, empirical software engineering.

Contents

1 Introduction **1**
1.1 Context: Software Maintenance and Evolution 1
1.2 Motivation 3
1.3 Problem Statement and Thesis 4
1.4 Contributions 6
1.5 Publications 7
1.6 Organisation of the Thesis 8

2 Background **9**
2.1 Studied Design Smells 9
2.2 Studied Systems 13
2.3 Faults and Fault-fixes 14
2.4 Support Vector Machines 16
2.5 Statistical Techniques for Analysis 18
2.5.1 Multivariate Regression Analyses 18
2.5.2 Mann-Whitney U-test 19
2.5.3 t-Test 20
2.5.4 Cliff d effect size 20
2.5.5 KruskalWallis Analysis 20

3 Related Work **21**
3.1 Design Smells Definition and Impact. 21
3.2 Design Smells Detection. 24
3.3 Summary 26

4 Evidence of the Impact of Design smells on Comprehension **27**
4.1 Context and Problem 28
4.2 Study Design 30
4.2.1 Research Questions 30

4.2.2 Objects and Subjects of the Study . . . 31
4.2.3 Independent and Mitigating Variables . . . 33
4.2.4 Dependent Variables . . . 34
4.2.5 Study Design and Procedure . . . 34
4.2.6 Analysis Method . . . 37
4.3 Study Results . . . 37
4.3.1 Impact of Two Occurrences of the Blob on Comprehension . . . 37
4.3.2 Impact of Two Occurrences of the Spaghetti code on Comprehension . . 38
4.4 Discussion . . . 39
4.4.1 Interpretation of the Results . . . 40
4.4.2 Impact of the Mitigating Variables . . . 41
4.5 Threats to Validity . . . 42
4.5.1 Construct Validity . . . 42
4.5.2 Internal Validity . . . 42
4.5.3 Conclusion Validity . . . 43
4.5.4 Reliability Validity . . . 43
4.5.5 External Validity . . . 43
4.6 Summary . . . 44

5 Evidence of the Impact of Design Smells on Fault-fixing Efforts 46

5.1 Context and Problem . . . 47
5.2 Study Design . . . 49
5.2.1 Research Questions . . . 49
5.2.2 Independent Variables . . . 52
5.2.3 Dependent Variables . . . 53
5.2.4 Analysis Method . . . 55
5.3 Study Results . . . 57
5.3.1 Relation between Design Smells and Faults . . . 58
5.3.2 Relation between Design Smells and the Duration of Fixing Periods . . . 60
5.3.3 Relation between Design Smells and the Number of Impacted Elements . 61
5.3.4 Relation between Design Smells and Fault-fixing Entropy . . . 62
5.3.5 Relation between Fault-fixing and the Number of Occurrences of Design Smells . . . 64
5.4 Discussion . . . 65
5.4.1 Statistically-significant Results . . . 65
5.4.2 Non-statistically-significant Results . . . 68
5.4.3 General Discussions . . . 69
5.5 Threats to validity . . . 69
5.5.1 Construct validity . . . 69

5.5.2 Internal Validity . . . 70
5.5.3 Conclusion Validity . . . 70
5.5.4 Reliability Validity . . . 70
5.5.5 External Validity . . . 70
5.6 Summary . . . 70

6 SMURF: A SVM-based, Incremental Design Smells Detection Approach 73
6.1 Context and Problem . . . 74
6.2 SMURF Process . . . 76
6.3 Empirical Study . . . 78
6.3.1 Research Questions . . . 78
6.3.2 Objects . . . 80
6.3.3 Subjects . . . 80
6.3.4 Data Collection and Oracles . . . 81
6.3.5 Analysis Methods . . . 82
6.4 Study Results . . . 83
6.4.1 Accuracy of SMURF Compared to That of DETEX . . . 83
6.4.2 Accuracy of SMURF Compared to That of BDTEX . . . 86
6.4.3 Accuracy of SMURF in Inter-systems Configurations . . . 87
6.4.4 Accuracy of SMURF using Users' Feedback . . . 88
6.5 Discussion . . . 88
6.5.1 SMURF vs. DETEX . . . 88
6.5.2 SMURF vs. BDTEX . . . 90
6.5.3 Other Design smells/Systems . . . 90
6.5.4 Practitioners' Feedback . . . 91
6.5.5 Software Evolution . . . 91
6.5.6 SMURF and Practitioners . . . 92
6.6 Threats to Validity . . . 92
6.7 Summary . . . 93

7 Conclusion 94
7.1 Conclusions . . . 94
7.2 Future Work . . . 96

A Definitions of Metrics 107

B Specification of Code Smells and Antipatterns 111
B.1 Detailed Definitions of the code Smells . . . 111
B.2 Detailed Definitions of the Antipatterns . . . 113

C Example of Comprehension Questions **116**

D TLX Rating Scale Definition **117**

E Post-mortem Questionnaire **118**

F Eclipse Tutorial **119**

List of Figures

2.1 Participation of classes in design smells in the analyzed releases 11
2.2 The optimal hyperplane and the maximal margins with the support vectors 16

4.1 Graphical representations of the collected data 45

6.1 SMURF process overview 77
6.2 Precision and recall of SMURF and DETEX on different subset size (Blob and Xerces) 84
6.3 Precision and recall of SMURF and DETEX on different subset size (Spaghetti Code and Xerces) 85
6.4 Trends in the increase of precision and recall when decreasing the probabilty of being an design smell for Blob and Xerces 86
6.5 Trends in the increase of precision and recall when decreasing the probabilty of being an design smell for Spaguetti Code and Xerces 87
6.6 Trends in the increase of precision and recall when integrating incremental feedback 89

List of Tables

2.1 Distribution of design smells in the analysed releases 12
2.2 Characteristics of the studied systems 14

4.1 Object Systems 32
4.2 Experimental design 34
4.3 Mann-Whitney tests and Cliff's *d* results for Experiment 1 37
4.4 Mann-Whitney tests and Cliff's *d* results for Experiment 2 38
4.5 Summary of the statistically-significant results 40
4.6 Kruskal-Wallis *p*-values of the impact of knowledge levels (AP = Design smell) . 41

5.1 Mann-Whitney tests and Cliff's *d* results for the number of occurrences of design smells in classes with faults compared to classes without faults across all versions 58
5.2 Mann-Whitney tests and Cliff's *d* results for the number of occurrences of design smells in classes with faults compared to classes without faults per version 59
5.3 Mann-Whitney tests and Cliff's *d* results for the duration of the fixing periods . . 60
5.4 Kinds of design smells significantly correlated with the duration of the fixing periods 61
5.5 Mann-Whitney tests and Cliff's *d* results for the number of elements impacted by fault-fixes 61
5.6 Kinds of design smells significantly correlated with the number of elements impacted by fault-fixes 62
5.7 Mann-Whitney tests and Cliff's *d* results for fault-fixing entropy 63
5.8 Kinds of design smells significantly correlated with the fault-fixing entropy . . . 63
5.9 Mann-Whitney results for the number of occurrences of design smells in classes before and after fault-fixes 64
5.10 Mann-Whitney results for the number of occurrences of design smells in classes before and after fault-fixes 65
5.11 Summary of the statistically-significant results of our study 72

6.1 Description of the objects of the study 80

6.2 Precision of SMURF in inter-systems 87
6.3 Recall of SMURF in inter-systems 88

List of Acronyms

BBN	*Bayesian Belief Network*
BDTEX	*Bayesian Detection EXpert*
CBO	*Coupling Between Objects*
CVS	*Concurrent Versions System*
DECOR	*DEtection&CORrection*
GQM	*Goal Question Metric*
IDE	*Integrated Development Environment*
IRS	*Issues Reporting System*
J2EE	*Java 2 Platform, Enterprise Edition*
EJB	*Enterprise JavaBeans*
NAS	*Number of Associations*
OO	*Object Oriented*
PADL	*Pattern and Abstract-level Description Language*
POM	*Primitives, Operators, Metrics*
PTIDEJ	*Pattern Trace Identification, Detection, Enhancement in Java*
UML	*Unified Modeling Language*
SMURF	*Support vector Machines for design smells detection Using Relevant Feedbacks*
SVM	*Support Vector Machines*
SVN	*apache SubVersioN*

To my parents, and sisters

To my wife and kids

Acknowledgements

Let us be grateful to people who make us happy;
they are the charming gardeners who make our souls blossom.
Marcel Proust

This Ph.D. study would not have been possible without the assistance and support of many people. I would like to tell them in these few lines the expression of my deep gratitude.

First of all, I would like to thank my supervisors, Esma Aïmeur and Yann-Gaël Guéhéneuc. I express my gratitude to Prof. Esma Aïmeur for agreeing to lead this work and the sense of engagement in the work that she inspired me. She taught me the rigor in the writing and presentation of research work. I want to express my deep gratitude and appreciation to Prof. Yann-Gaël Guéhéneuc, for trusting me in the choice of this research topic. Especially, I am grateful for his many constructive and clear advice on how to approach research problems and the need to be persistent and realistic to achieve the goal. Many thanks to him for showing me the joy of research, for shaping my path to research by guiding me with his extensive knowledge step by step, for teaching me how to write, for creating an enjoyable working environment and listening to my personal problems all the time, for his unending encouragement and support, and for becoming, more than a supervisor, a friend.

My thankful admiration goes to Prof. Giuliano Antoniol, for all the essential advices and valuable comments that I received from him.

Special thanks go to Prof. Ahmed E. Hassan for enthusiastically accepting to be my external examiner and for reviewing my thesis. I thank Prof. Bruno Dufour for accepting to be on my committee and for reviewing my thesis.

A Special thanks goes to Prof. Philippe Langlais, who have enthusiastically accepted to chair my doctoral committee.

Many thanks go to Foutse Khomh, Marwen Abbes, Nasir Ali and Aminata Sabané for all the interesting technical discussion, for their honest criticism, and suggestions, for the intensive but

interesting time during paper writing. Thanks to my fellow doctoral students from the Ptidej and Soccer labs for their valuable comments and suggestions.

I am grateful to my parents, especially my mother, for their love, for their unconditional support despite all the difficulties. Many thanks to my wife, Djénéba Fofana and my kids, Ibrahim, Mohamed, Ammar and, Youssouf, for their love and understanding during all these years. I would like also to thank them for their patience during my periods of absence for research purposes. I love you.

This research was funded by Canadian International Development Agency (CIDA). I am grateful to them for supporting my Ph.D. and giving me the opportunity to come and study in Canada.

—Abdou Maiga

Chapter 1

Introduction

In this chapter, we present the context of our research work, which is agile software development. We state the relationship between design smells and comprehension and change. We present the problems and our thesis saying that there is a strong relationship between the understanding of software systems, the effort required for their change, and the occurrences of design smells, in an agile software development context. We present our approach, SMURF, to detect occurrences of design smells, taking into account the practitioners' feedback to improve the quality of the software systems.

Contents

1.1 Context: Software Maintenance and Evolution **1**
1.2 Motivation **3**
1.3 Problem Statement and Thesis **4**
1.4 Contributions **6**
1.5 Publications **7**
1.6 Organisation of the Thesis **8**

1.1 Context: Software Maintenance and Evolution

Software systems are prevalent in all areas of societies including the most sensitive as medicine, economics. They therefore play a central role in our societies. In this context, the quality of software systems becomes crucial. The development of software involves several steps. These steps include the specification of requirements, design, implementation, testing, deployment, and maintenance. Software maintenance is defined by the IEEE Standard 1219 [IEEE, 1999] as: the

modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. Maintenance activities have become expensive nowadays. They reach more than 70% of the overall costs of software development [Pressman, 2001]. Bennett and Rajlich [2000] recall the categories of maintenance:

- Adaptive - changes in the software environment;
- Perfective - new user requirements;
- Corrective - fixing errors;
- Preventive - prevent problems in the future.

They showed that around 75% of the maintenance effort concern adaptive and perfective maintenance while 21% of the maintenance effort is spent in error correction. The most important kinds of maintenance are adaptive and perfective maintenance. Developers continuously evolve their systems to implement new functionality as well as to fix faults. For example software evolution is central to agile software development [Beck *et al.*, 2001], which is a set of collaborative, iterative, and incremental development methods. Agile methods are based on four common values, which are the team, the working software, the customer collaboration, and responding to change [Beck *et al.*, 2001]. Martin *et al.* [2003] summarise the characteristics of agile methods as follows: "Agile Methods are:

- Iterative: the team delivers a full system at the very beginning and then changes the functionality of each subsystem with each new release;
- Incremental: the system as specified in the requirements is partitioned into small subsystems by functionality. New functionality is added with each new release;
- Self-organizing: the team has the autonomy to organise itself to best complete the work items;
- Emergent: technology and requirements are "allowed" to emerge through the product development cycle."

Since its introduction, agile software development methods gained in popularity and are the most used software development methods in industry. Many books and articles have been written to explain the methods and help practitioners to apply them in their projects [Martin, 2003]. For software maintenance and evolution activities, comprehension and change are important. The comprehension of a system is the degree to which its source code can easily be understood by developers [Ignatios *et al.*, 2003 ; Ignatios *et al.*, 2004 ; Khomh and Guéhéneuc, 2008].

Design patterns are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability, resilience to changes and, above all, maintainability [Gamma *et al.*, 1994]. Due to the time-to-market, lack of understanding, and their experience, developers cannot always follow standard designing and coding techniques, *i.e.*, design patterns [Fowler, 1999a]. Therefore, they introduced *antipatterns* and *code smells* in their systems. *Antipatterns* and *code smells* are poor solutions to recurring design or implementation problems [Webster, 1995]. In this thesis, as in previous research work [Moha *et al.*, 2009], we use the term "design smells" for both antipatterns and code smells. Design smells reflect developers' expertise. They are generally the result of misuse of the object-oriented paradigm and–or design patterns [Brown *et al.*, 1998]. Consequently, design smells creep up in systems. An example of design smell is the *Blob*, also called *God Class*, which is a large and complex class that centralises most of the behavior of a system. A Blob also depends on data stored in surrounding *data classes*. A Blob is characterised by a low cohesion and a large size. It prevents the use of polymorphism through inheritance [Moha *et al.*, 2010c]. Another example of design smell is the *MessageChain*, which occurs when the implementation of a functionality in one class requires a long chain of method invocations between objects from different classes [Moha *et al.*, 2010c]. Classes with message chains are more prone to faults than others [Khomh *et al.*, 2011a].

1.2 Motivation

Previous work showed the negative impact of design smells on software quality attributes and also on comprehension [Abbes *et al.*, 2011] and change [Khomh *et al.*, 2011a]. Software systems are always subject to modification to adapt or implement new requirements. Therefore, software systems evolve over time and the understandability of the source code is important to adapt, implement new requirements but also to fix fault. Researchers have performed empirical studies to show that design smells like Spaghetti Code combined with Blob create hurdles during program comprehension [Abbes *et al.*, 2011], software evolution, and maintenance activities [Khomh *et al.*, 2011a].

The importance of these two activities, comprehension and change, in the software maintenance and evolution activities, motivate us to gather quantitative evidences on the relation between design smells and comprehension, faults, and the developers' effort to fix the faults. Bringing these evidences is important for practitioners, *i.e.*, developers, testers, and managers, and will contribute to reduce maintenance costs and improve software quality. Indeed, the knowledge of the impact of design smells on comprehension and fault-fixing activities will help practitioners to take rational decisions about how dealing with design smells.

Thus, evidences of a negative impact of design smells on comprehension, fault-fixing activities, and on a relation between the fixing of faults and the number of occurrences of design smells will allow us to advise practitioners to be more cautious with design smells, *i.e.*, with their design decisions (or lack thereof). These evidences adding to previous results [Moha *et al.*, 2010c ; Abbes *et al.*, 2011 ; Khomh *et al.*, 2011a] could help practitioners justify the removal of design smells. Previous work [Moha *et al.*, 2009] discussed the importance for maintenance activities to automatically detect occurrences of design smells. We now link design smells and effort concretely. The tools proposed so far in the literature to detect design smells are not iterative and incremental. To fill this gap and contribute to the reduction of software maintenance costs, we provide SMURF, a tool based on a machine learning algorithm to automatically detect the occurrences of design smells iteratively and incrementally. SMURF takes into account the practitioners' feedback and allows iterative and incremental detection with higher accuracy. If developers independently detect design smells among few classes of their systems as there are developed and incrementally add more classes to the previous release, the correction or removing of those design smells will be less costly than in the whole system at the end of the development.

1.3 Problem Statement and Thesis

In the literature, design smells are conjectured to negatively impact comprehension and the effort to fix the faults. However, only few studies empirically investigated the impact of design smells on comprehension and change (through fault-fixing activities) as reported in Chapter 3.

Comprehension is the central element for the effectiveness of software maintenance and evolution [von Mayrhauser and Vans, 1995]: a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension: in other words, its change. Also the effort to fix a fault has an important effect on the costs of software maintenance in terms of time and money.

Our thesis is that it is possible to bring quantitative evidences of the impact of design smells on comprehension and fault-fixing effort and provide a tool for accurate incremental and iterative design smells detection.

Therefore, we tackle the three (3) following problems:

- The lack of evidence of the impact of design smells on comprehension;
- The lack of evidence of the impact of design smells on fault-fixing effort;
- The limitations of the previous design smells detection approaches.

Evidence of the Impact of Design Smells on Comprehension. In the literature, design smells are conjectured to negatively impact comprehension but few studies have empirically investigated this impact of design smells. In software maintenance and evolution context, it is important to investigate the impact of design smells on comprehension because this is one of the most important activities in such a context. To investigate this impact, we design and conduct two experiments with 59 subjects. In these experiments we assess the impact of the composition of two occurrences of Blob or two occurrences of Spaghetti Code on the performance of developers performing comprehension and change tasks. We analyse whether increasing the number of occurrences of some design smell, *i.e.*, two occurrences of the Blob or two occurrences of the Spaghetti Codes, in some systems impacts the comprehension.

We gather quantitative evidence of the impact of Blob and Spaghetti Code design smells on comprehension. We measure developers' performance using: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. The results of the two experiments show that two occurrences of Blob or Spaghetti Code design smells significantly impede developers' performance during comprehension and change tasks. The obtained results justify a posteriori previous research on the specification and detection of design smells. Software development teams should warn developers against high number of occurrences of design smells and recommend refactorings at each step of the development to remove them when possible.

Gathering these evidences of the relation between Blob and Spaghetti Code design smells and comprehension will help us to advise practitioners to be more cautious with design smells, *i.e.*, with their design decisions (or lack thereof), especially in agile software development context where comprehension is very important.

Evidence of the Impact of Design Smells on Fault-fixing Effort. When doing maintenance and evolution changes, developers face faults and must fix them continuously. Design smells are prevalent in systems [Khomh *et al.*, 2011a] and several studies [Abbes *et al.*, 2011 ; Khomh *et al.*, 2011a] have shown that they negatively impact software development and maintenance. Khomh *et al.* [Khomh *et al.*, 2011a] showed that classes participating in design smells are more prone to changes and faults than others. However, none of previous work empirically investigated the impact of design smells on the developers' effort to fix faults.

We gather quantitative evidence on the relation between design smells, faults, and the developers' effort to fix the faults. We study the impact of the presence of design smells on the effort required to fix faults, which we measure using three metrics: (1) the duration of the fixing period; (2) the number of fields and methods impacted by fault-fixes; and, (3) the entropy of the fault-fixes

in the source code. We conduct an empirical study with 12 design smells detected in 54 releases of four systems: ArgoUML, Eclipse, Mylyn, and Rhino. Our results show that the duration of the fixing period is longer for faults involving classes with design smells. Also, fixing faults in classes with design smells impacts more files, more fields, and more methods. We also observed that after a fault is fixed, the number of occurrences of design smells in the classes involved in the fault decreases. Understanding the impact of design smells on development effort is important to help development teams better assess and forecast the impact of their design decisions and therefore lead their effort to improve the quality of their software systems. Development teams should monitor and remove design smells from their software systems because they are likely to increase the change efforts.

Incremental Design Smells Detection Approach. Due to the impact of design smells shown in our two first contributions, it is important to detect design smells incrementally and iteratively to help developers to remove them early. If developers independently detect design smells among few classes of their systems as they are developed, and remove them before adding more classes to produce the next release, the maintenance costs will be reduced as ccomprehension will be easier and the effort to fix faults reduced. Another factor that participates to reduce the cost is that the removing of design smells at this stage will be easier and less expensive than in the whole system at the end of the development process. Current design smells detection approaches as proposed by Marinescu [Marinescu, 2004a], Moha et *al.* [Moha *et al.*, 2009] and Alikacem et *al.* [Alikacem and Sahraoui, 2006] are not iterative and incremental and they required extensive knowledge of design smell specifications. Therefore we propose SMURF, a SVM-based, incremental and iterative design smells detection approach. SMURF is an approach based on a machine learning technique—support vector machines—and taking into account practitioners' feedback. Through an empirical study involving three systems and four design smells, we show that the accuracy of SMURF is greater than that of DETEX and BDTEX when detecting design smells occurrences. We also show that SMURF can be applied in both intra-system and inter-system configurations. Finally, we reported that SMURF accuracy improves when using practitioners' feedback.

1.4 Contributions

The main contributions of this thesis follow three axes:

1. To answer the problem of lack of evidence of the impact of design smells on comprehension, we conduct an empirical study of the impact of Blob and Spaghetti Code design smells on

comprehension. We found that the combination of two Blob or two Spaghetti Code design smells impedes significantly developers performance during comprehension and change.

2. To answer the problem of lack of evidence of the impact of design smells on fault-fixing effort, we conduct an empirical study on the evidence of the impact of design smells on fault-fixing efforts. We found that the duration of the fixing period is longer for faults involving classes with design smells. Also, fixing faults in classes with design smells impacts more files, more fields, and more methods. We also observed that after a fault is fixed, the number of occurrences of design smells in the classes involved in the fault decreases.

3. To answer the problem of the limitations of the previous design smells detection approaches, we propose an approach named SMURF to detect iteratively and incrementally design smells. We exploit the benefits of SVM to detect the occurrences of design smells while taking into account practitioners feedback.

1.5 Publications

This research work has been the object of the following publication:

Abdou Maiga, Nasir Ali, Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Nesa Asoudeh, Yvan Labiche, (2012) An Empirical Study of the Impact of Blob and Spaghetti Code Antipatterns On Program Comprehension, Journal of Empirical Software Engineering (EMSE) (submitted).

Abdou Maiga, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Esma Aïmeur, (2012) Evidences of the Impact of Design Smells on Fault-fixes Efforts, Journal of Empirical Software Engineering (EMSE) (submitted).

Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Esma Aïmeur, (2012) Support Vector Machines for Antipattern Detection, Proceedings of the 27th IEEE International Conference on Automated Software Engineering (ASE'12), 3-7 September 2012, IEEE Computer Society Press. Short paper.

Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, Esma Aïmeur, (2012) SMURF: A SVM-based, Incremental Antipattern Detection Approach, In Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), October 15-18, Kingston, Ontario (Canada). IEEE Computer Society Press.

1.6 Organisation of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2 (p. 9) provides background materials required to understand this thesis. It presents the studied objects and systems used in our empirical studies. It also recalls the machine learning technique and statistical techniques used in our analysis.

Chapter 3 (p. 21) provides a brief description of the state-of-the-art of design smells definition and detection approaches, the link between design smells and evolution, developers, changes and faults

Chapter 4 (p. 27) explores the study of the impact of design smells on comprehension

Chapter 5 (p. 46) explores the study of the impact of design smells on fault-fixing effort.

Chapter 6 (p. 73) presents an iterative and incremental design smells detection approach, SMURF.

Chapter 7 (p. 94) presents the conclusions of this thesis and outlines some directions of future works.

Appendix A (p. 107) presents the definitions of the metrics used in this thesis.

Appendix B (p. 111) presents the complete list of code smells and antipatterns considered in this thesis with their definitions.

Appendix C (p. 116) presents some examples of comprehension questions submitted to the subjects.

Appendix D (p. 117) presents the definition of TLX Rating Scale.

Appendix E (p. 118) presents the post-mortem questionnaire.

Appendix F (p. 119) presents the Eclipse tutorial.

Chapter 2

Background

In the following, we provide background materials required to understand this thesis. We provide definitions of design smells, faults, and fault fixes and justify our choice of the subject systems and studied smells. We also recall the machine learning and statistical techniques used in our analyses.

Contents

2.1 Studied Design Smells **9**

2.2 Studied Systems **13**

2.3 Faults and Fault-fixes **14**

2.4 Support Vector Machines **16**

2.5 Statistical Techniques for Analysis **18**

2.5.1 Multivariate Regression Analyses 18

2.5.2 Mann-Whitney U-test 19

2.5.3 t-Test 20

2.5.4 Cliff d effect size 20

2.5.5 KruskalWallis Analysis 20

2.1 Studied Design Smells

In this thesis, we consider the well known 12 design smells from Brown's and Fowlers' books [Brown *et al.*, 1998 ; Fowler, 1999a]. We briefly describe these 12 design smells in the following. A complete list and description of design smells is provided in appendix B.

1. AntiSingleton (ATS): A class that provides mutable class variables, which, consequently, act as global variables.
2. Blob (BLB): A class that is too large and not cohesive enough. This class centralizes most of the processing, takes most of the decisions, and is associated to data classes.
3. ClassDataShouldBePrivate (CDSBP): A class that exposes its fields, thus violating the principle of encapsulation.
4. ComplexClass (CC): A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
5. LargeClass (LC): A class that has grown too large in term of LOCs.
6. LazyClass (LYC): A class that has few fields and methods (with little complexity).
7. LongMethod (LM): A class that has a method that is overly long, in term of LOCs.
8. LongParameterList (LPL): A class that has (at least) one method with a long list of parameters with respect to the average number of parameters per methods in the system.
9. MessageChain (MC): A class that uses a long chain of method invocations to implement (at least) one of its functionality.
10. RefusedParentBequest (RPB): A class that redefines inherited methods using empty bodies, thus breaking polymorphism.
11. SpeculativeGenerality (SG): A class that is defined as abstract but that has very few children. Her children do use its methods.
12. SwissArmyKnife (SAK): A class whose methods can be divided in disjunct sets of many methods, *i.e.*, providing different, unrelated functionalities.

We choose these design smells because they have been used in previous studies, *eg.*, [Khomh *et al.*, 2009b ; Moha *et al.*, 2009]. They are representative of design and implementation problems with data, complexity, size, and the features provided by classes [Khomh *et al.*, 2011a].

We also choose these 12 design smells because the percentages of classes participating in these design smells are not negligible in the studied systems. Figure 2.1 shows that these percentages vary across releases of the studied systems and that they are always higher than 45%, with peaks as high as 80%.

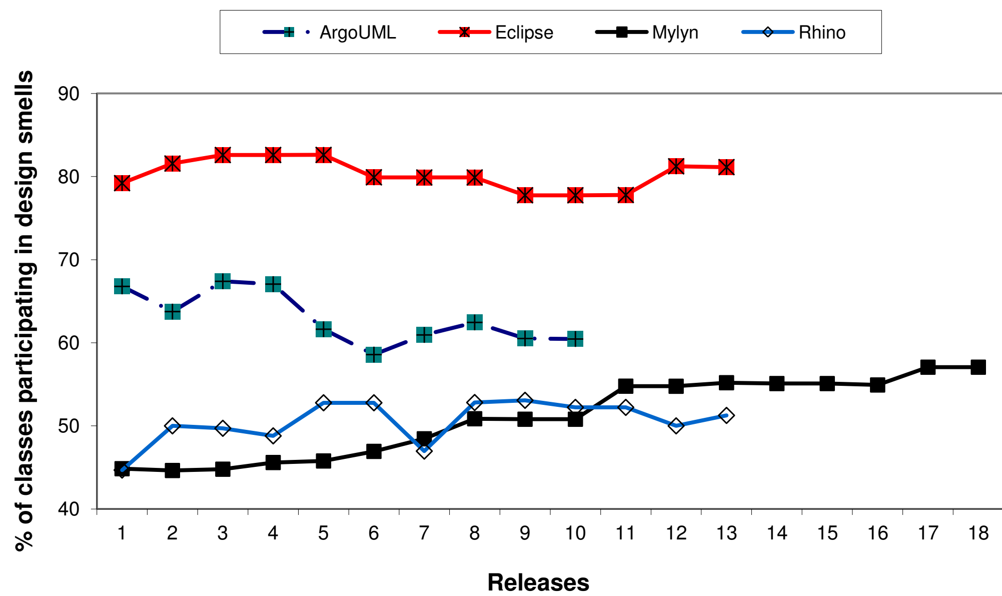


Figure 2.1 – Participation of classes in design smells in the analyzed releases.

Table 2.1 – Distribution of design smells in the analysed releases.

Design smells	Number of Design Smells in First and Last Releases (in parentheses, the percentages of participating classes)			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	352 (44.44)–3 (0.16)	330 (7.10)–1784 (10.39)	4 (0.25)–127 (4.60)	16 (17.98)–1 (0.37)
Blob	26 (3.28)–116 (6.30)	600 (12.91)–2,194 (12.78)	40 (2.46)–93 (3.37)	0 (0)–0 (0)
CDSBP	136 (17.17)–51 (2.77)	382 (8.22)–2,285 (13.31)	61 (3.75)–183 (6.63)	4 (4.49)–17 (6.30)
ComplexClass	42 (5.30)–103 (5.59)	511 (11.00)–2,125 (12.38)	29 (1.78)–72 (2.61)	6 (6.74)–14 (5.56)
LargeClass	56 (7.07)–166 (9.02)	1 (0.02)–8 (0.05)	43 (2.65)–99 (3.58)	9 (10.11)–19 (7.04)
LazyClass	16 (2.02)–44 (2.39)	2,403 (51.71)–8,561 (49.87)	2 (0.12)–18 (0.65)	4 (4.49)–9 (3.33)
LongMethod	172 (21.72)–348 (18.90)	2,372 (51.04)–7,956 (46.34)	134 (8.25)–349 (12.64)	14 (15.73)–35 (12.96)
LPL	195 (24.62)–300 (16.30)	1,087 (23.39)–3,233 (18.83)	43 (2.65)–95 (3.44)	9 (10.11)–8 (2.96)
MessageChain	79 (9.97)–166 (9.02)	1,043 (22.44)–3,041 (17.71)	70 (4.31)–181 (6.55)	20 (22.47)–66 (24.44)
RPB	105 (13.26)–574 (31.18)	397 (8.54)–2,582 (15.04)	45 (2.77)–290 (10.50)	5 (5.62)–11 (4.07)
SpaghettiCode	9 (1.14)–22 (1.20)	2 (0.04)–1 (0.01)	12 (0.74)–39 (1.41)	0 (0.00)–2 (0.74)
SG	0 (0.00)–0 (0.00)	54 (1.16)–228 (1.33)	0 (0.00)–0 (0.00)	0 (0.00)–0 (0.00)
SwissArmyKnife	0 (0.00)–0 (0.00)	67 (1.44)–96 (0.56)	1 (0.06)–0 (0.00)	0 (0.00)–0 (0.00)

Table 2.1 reports the distribution of the studied design smells. A cell in the table reports, on the left side of the dash (respectively, on its right), the number of classes in the first release of a given system (respectively, its last) with a particular design smell, followed by their percentages with respect to the total numbers of classes. For example, the cell at the intersection of the ArgoUML column and the AntiSingleton row reports that in the first release of ArgoUML, 352 classes were AntiSingleton, representing 44.44% of the total number of its classes, while in the last release, only three classes were AntiSingleton, representing 0.16% of the total number of its classes. Percentages go as high as 51.71% of classes participating in LazyClass in the first release of Eclipse.

Khomh *et al.* [Khomh *et al.*, 2011a] also report that classes participating in design smells participate, in average to a minimum of, two distinct design smells in ArgoUML, three in Eclipse, two in Mylyn, and two in Rhino and, to a maximum of, nine unique design smells in ArgoUML, 24 design smells in Eclipse, 7 design smells in Mylyn, and 7 design smells in Rhino [Khomh *et al.*, 2009b]. Thus, enough occurrences of the 12 chosen design smells are present, both in raw numbers and in percentages, in the studied systems to perform statistical analyses.

We identify the occurrences of design smells in systems using DECOR (Defect dEtection for CORrection) [Moha *et al.*, 2010b].

DECOR is an approach based on the automatic generation of detection algorithms from rule cards. DECOR proposes the descriptions of different design smells and provides algorithms and a framework called DetEx. DetEx convert design smells descriptions automatically into detection algorithms and extract different occurrences of design smells. We apply DetEx in three steps: first, we reuse/define a rule card describing a smell through a domain analysis of the literature on the smell. From the rule card, we generate a detection algorithm. Finally, we apply the detection algorithm on a model of a system to detect the different occurrences of the design smell in this system.

2.2 Studied Systems

The subject systems of our studies are four open-source Java programs: ArgoUML, Eclipse, Mylyn, and Rhino. ArgoUML is an open-source UML-based system design tool. Eclipse is an open-source integrated development environment. Mylyn is a plug-in for Eclipse, which aims at reducing information overload and making developers' multi-tasking easier. Rhino is an open-source implementation of a JavaScript interpreter.

Table 2.2 – Characteristics of the studied systems.

System	Versions (Number)	LOCs	Number of classes
ArgoUML	0.10.1-0.26.2 (10)	128,585-316,971	792-1,841
An open-source UML-based system design tool			
Eclipse	1.0-3.3.1 (13)	781,480-3,756,164	4,647-17,167
An open-source integrated development environment			
Mylyn	1.0.1-3.1.1 (18)	207,436-276,401	1,625-2,762
A plug-in for Eclipse			
Rhino	1.4R3-1.6R6 (13)	30,748-79,406	89-270
An open-source implementation of a JavaScript interpreter			

These four systems are well known, belong to different domains, and have different architectures. They have been used in previous studies, *eg.*, [Khomh *et al.*, 2011a ; Moha *et al.*, 2009 ; Khomh *et al.*, 2011b]. We report some descriptive statistics of the four systems in Table 2.2.

2.3 Faults and Fault-fixes

Developers usually document faults in a system to warn their community of pending issues with the system functionalities. They use issue-tracking systems to post the descriptions of faults and the descriptions on their fixes. A common issue-tracking system, used for the four studied systems, is Bugzilla. We analyse Bugzilla issues as did previous work [Zaman *et al.*, 2011] to identify faults in the studied systems. Developers also use, in addition to issue-tracking systems, version-control systems, such as CVS or SVN, to share the source code in which they have integrated their fault fixes. Therefore, we use these two sources of information to identify fault fixes performed by developers to fix faults in the four studied systems: issue-tracking systems and version-control systems.

First, we use a mapping between faults and fault fixes provided by Khomh *et al.* [Khomh *et al.*, 2011a], who identified changes to classes in the studied systems to fix faults by looking at the commits logs of the systems in their version-control systems and by relating the changes to faulty classes using the IDs of the faults in the commit logs [Fischer *et al.*, 2003]. We are interested in particular to faults which have been set to "Fixed" or "Closed". In the particular case of Rhino, we use a manually-validated, publicly-available mapping between faults and commits [Eaddy *et al.*, 2008]. The obtained mappings allow us to relate faults and classes and, using the occurrences of design smells provided by DECOR, we can relate classes with design smells and, thus, relate faults to design smellS by transitivity.

Second, we also consider the patches attached to faults in the issue-tracking systems. When developers fix a fault, they may upload in their issue-tracking system a patch, which is a piece of code describing the modifications to the code base to fix the fault. Patches typically identify the lines of codes and the related classes and methods that the developers modified to fix the fault. We use these patches to relate faults to occurrences of design smells, on the one hand, with the characteristics of the particular changes performed by developers to fix the faults, on the other hand.

Using these two sources of information, we say that a fault t involves a class c when the developers modified c to fix the fault t. We call r the release against which the fault was reported by the developers, *i.e.*, a patch was submitted against release r and–or the source code of release r include code fixing the fault t. Thus, using the releases against which a fault is reported and fixed, we can collect the following pieces of data:

- The duration of the fixing period D_f in term of number of days. We compute D_f using the date of the introduction of the description of the fault in the issue-tracking system and that of the introduction of the patch and–or the commit of source code commented as fixing the fault.
- The number Nbr_{Elts} of any field and–or method added/removed when the fix is introduced by developers. We compute Nbr_{Elts} by parsing the patch diff files to count the number of added/removed fields $N_{f,t}$ and the number of added/removed method $N_{m,t}$.
- The entropy of the fault fixes. The entropy of the fault fixes represents the complexity of the fault-fixes [Zaman *et al.*, 2011], *i.e.*, the dispersion of the fault-fixing changes in different classes to fix the fault. To compute this entropy, we consider:
  - n, the number of files modified to fix a fault;
  - l_i, the number of lines of code changed (added/deleted) by the changes for $file_i$;
  - $l = \sum_{i=1}^{n}(l_i)$, the total number of lines of code changed for all files.

  We then use the normalised Shannon Entropy as:

$$H_n(P) = -\sum_{i=1}^{n}(\frac{l_i}{l} \times log_n \frac{l_i}{l})$$

  where $\frac{l_i}{l} \geq 0; i = 1, 2, \ldots, n$ and $\sum_{i=1}^{n} \frac{l_i}{l} = 1$.

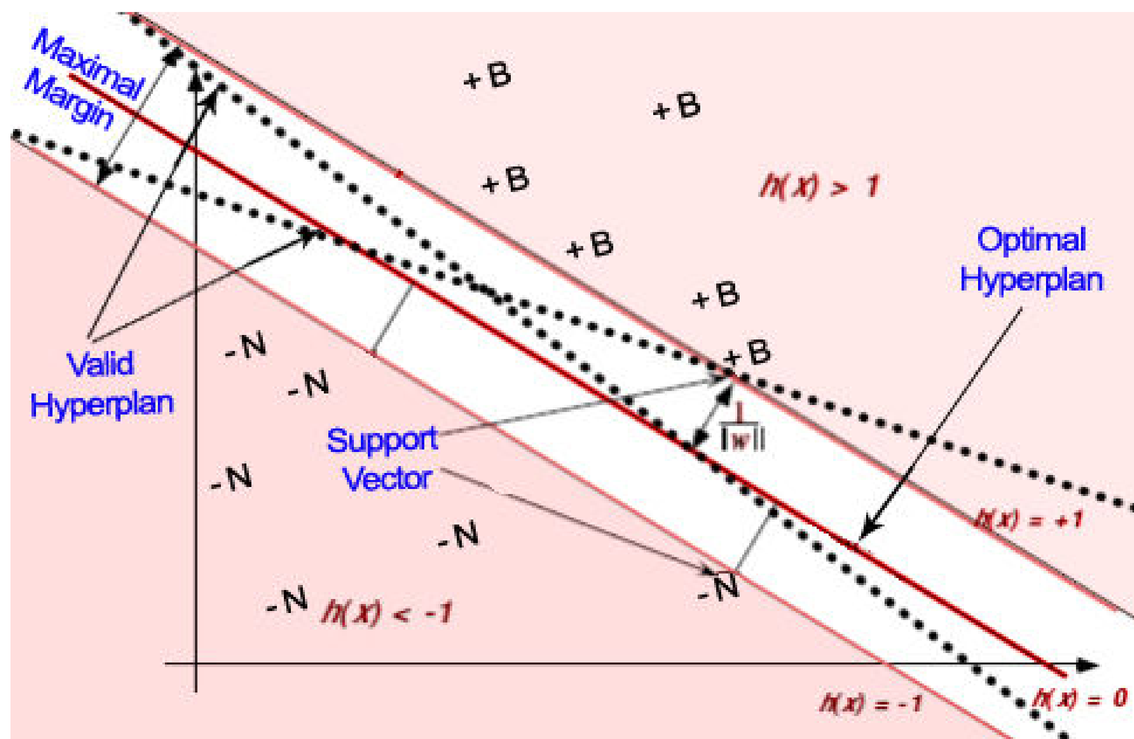


Figure 2.2 – The optimal hyperplane and the maximal margins with the support vectors.

The duration of the fixing period D_f, The number Nbr_{Elts}, and the entropy of the fault fixes $H_n(P)$, constitute the measures used in the present study to relate design smells to developers' fault-fixing efforts.

2.4 Support Vector Machines

SVM is a set of techniques based on statistical theory of supervised learning introduced by Vapnik [Cortes and Vapnik, 1995]. It relies on the existence of a linear classifier in an appropriate space and uses a set of training data to train the parameters of the classifier. SVM is based on the use of functions called kernel, which allows an optimal separation of data into two categories by a hyperplane. Figure 2.2 shows a hyperplane that separates two sets of points.

Assuming some training data $x = x_1 \ldots x_n$ and their labels $y = y_1 \ldots y_n$ that are vectors in the space $\mathbb{R}^n$, which has a dot product, where $y_i \in \{-1, 1\}$, SVM builds a function f to assign for a given x_i, the label y_i. The function f uses an hyperplane to assign the labels by separating the hyperspace in two. There are many valid hyperplanes but SVM can compute the optimal

hyperplane, which intuitively passes in the "middle"' of the two sets of data. Formally, the SVM finds the hyperplane for which the minimum distance to the training examples is maximal. The margins are the distances between the hyperplane and the closest examples. These margins are called support vectors, as shown in Figure 2.2. The optimal separating hyperplane is the one that maximises the margin. A hyperplane is defined as a set of points x satisfying $h(x) = w \cdot x + b = 0$ where $\cdot$ is the dot product; w is the normal vector to the hyperplane; and the parameter $\frac{b}{||w||}$ is the perpendicular distance from the hyperplane to the origin. Therefore, there are two support vectors: x^{+}_{marge} for the support vector (positive border) near to positive class C_{+} and x^{-}_{marge} for the support vector (negative border) near to negative class C_{-}. The following equation describes the support support vectors:

$$\begin{cases} w \cdot x^{+}_{marge} + b = 1 \\ w \cdot x^{-}_{marge} + b = -1 \end{cases}$$

The distance (margin) between the two support vectors is calculated by subtracting their equations: $Margin = \frac{2}{||w||}$. Maximising the margin is equivalent to minimizing $||w||$. To prevent data points from falling into the margin, the following constraint is applied:

$$\forall i, \begin{cases} w \cdot x_i + b \geq 1 \forall x_i = 1, \ldots, p \\ w \cdot x_i + b \leq -1 \forall x_i = 1, \ldots, p \end{cases}$$

The computation of the maximum margin and the search procedure of separating hyperplane can often only be resolved linearly through separable discrimination problems. SVM suggests reconsidering the problem of the lack of linear separator in a higher dimensional space (possibly infinite dimensional). In this new space, it is likely that there is a linear separator. Formally, a non-linear transformation φ can be applied to the input vectors x. The arrival space $\varphi(X)$ is called the space of re-description. In this space, the hyperplane equation becomes: $h(x) = w \cdot \varphi(x) + b$, which verifies $y_k \cdot h(x_k) > 0$, for all the points x_k in the training dataset, *i.e.*, the separating hyperplane in the space of re-description. The following optimization problem is obtained:

$$\begin{aligned} \text{Maximise } \widetilde{L}(\alpha) = \sum_{k=1}^{p} \alpha_k - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi(x_j) \\ \text{subject to } \alpha_i \geq 0 \\ \text{and } \sum_{k=1}^{p} \alpha_k y_k = 0 \end{aligned} \tag{2.1}$$

The problem with this formulation is that it implies a dot product between vectors in the space of re-description, with high dimensions, which is costly in terms of computations. To resolve

this problem, a kernel function K is used, which satisfies: $K(x_i, x_j) = \phi(x_i)^T \cdot \phi(x_j)$. Hence, the expression of the hyperplane becomes:

$$h(x) = \sum_{k=1}^{p} \alpha_k^* y_k K(x_k, x) + w_0 \tag{2.2}$$

The kernel function allows to perform the computations in the original space, which is much less expensive than high-dimensional dot product. Further, the transformation φ does not need to be known explicitly, only the kernel function is involved in the computations. The kernel function may lead to complex transformations and even re-description of spaces of infinite dimension [John Shawe-Taylor, 2000].

2.5 Statistical Techniques for Analysis

The analysis of the results of our experiments in this thesis involve classifiers, population tests, and correlational analyses. In the following, we discuss these statistical techniques.

2.5.1 Multivariate Regression Analyses

In our thesis, we use multivariate regression analyses to analyse the relation between particular kinds of design smells and a study phenomenon. To determine the weight of each kind of design smells in numbers, we use multivariate regression analyses because one of the uses of regression analyses is to understand which, among some independent variables, *i.e.*, the different kinds of design smells, are related to the dependent variables, *i.e.*, the durations of fixing periods, the number of elements impacted by fault-fixes, and the entropy of fault-fixes.

The general linear regression model with p independent variables is based on the formula:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i$$

where:

- x_{ij} are characteristics describing the modelled phenomenon, in our case, the number of each kind j of design smells involved in the fixing of fault i;
- β_j are the model coefficients.

Based on the p-values of each variable x_{i1}, x_{i2}, …, x_{ip}, we reject the null hypotheses H_{02K}, H_{03K}, and H_{04K}, if the p-value of the variable x_{ip} that describes a particular kind of design smells

is less than α. If p-value $< \alpha$ then we assume that the particular design smell impacts more the developers' fault-fixing effort than others.

2.5.2 Mann-Whitney U-test

In this thesis, we compute the Mann-Whitney U (also called Mann-Whitney-Wilcoxon or Wilcoxon Rank-Sum Test) test to compare two groups of observation.

The Mann-Whitney U-test is a non-parametric test. We use the Mann-Whitney U-test because, as a non-parametric test, it does not make any assumption on the underlying distributions. This test can be used for very small group (The minimum size of each group is between 4 and 8 according to the authors). The Mann-Whitney U is used to test the null hypothesis that two groups come from the same distribution or alternatively, whether observations in one group tend to be larger than observations in the other group.

For example, we want to compare the average time to answer a question on a system containing occurrences of design smells (Group G_x) versus the time average to answer a question on a system without occurrences of design smells (Group G_y). We will test the null hypothesis that the average time of the two groups (G_x and G_y) are similar, or alternatively, whether the difference between the average time of the two groups is statistically significant different and the average time to answer a question on a system containing occurrences of design smells tend to be larger than the average time to answer a question on a system without occurrences of design smells. In the Mann-Whitney U-test, for each time x_i, we compare x_i to every y_j. The number of possible comparisons that we can make is $n_x * n_y$. We consider the following :

- n_x: number of subjects who answer question from Group G_x and we note their time $\{x_1, x_2, \ldots x_n\}$.
- n_y: number of subjects who answer question from Group G_y and we note their time $\{y_1, y_2, \ldots y_n\}$.
- U_x: the number of times an x_i from Group G_x is greater than a y_j from Group G_x
- U_y: the number of times an x_i from Group G_x is smaller than a y_j from Group G_x
- $U = min(U_x, U_y)$

In order to reject or accept the null hypothesis, we compare the value U with the value given in the Tables of Critical Values for the Mann-Whitney U-test. Using this tables for Mann-Whitney U-test we get a p-value. If p-value $< \alpha$, we reject the null hypothesis and alternatively conclude that the average time to answer a question on a system containing occurrences of design smells is larger than the average time to answer a question on a system without occurrences of design smells. In this thesis, results are intended as statistically significant at $\alpha = 0.05$

2.5.3 t-Test

In addition to performing non-parametric tests, we also test our hypotheses with the (parametric) t-test. Performing the t-test is of practical interest to estimate the differences between the mean of a phenomenon under study (*eg.*, the difference of the number of design smells in classes with and without faults). A paired t-Test is used to test the null hypothesis that two groups have the same mean for an observed phenomenon.

Other than testing the hypotheses with Mann-Whitney U-test and t-Test, we also estimate the magnitude of the difference in means between the two groups. We use the non-parametric effect size measure Cliff's d to capture this magnitude.

2.5.4 Cliff d effect size

We use the non-parametric effect size measure Cliff's d, which indicates the magnitude of the effect size of the treatment on the dependent variable. Cliff's d is defined as:

$$d = \frac{\#(y_i > x_j) - \#(y_i < x_j)}{n_y.n_x} = \frac{\sum_i \sum_j d_{ij}}{n_y.n_x}$$

where $d_{ij} = sign(y_i - x_j)$ and $\#(y_i > x_j)$ the number of comparisons between observations in the two groups for which the Group i observation is larger than the Group j observation.

The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [Grissom and Kim, 2005].

2.5.5 KruskalWallis Analysis

We use KruskalWallis one-way analysis of variance by ranks to analyse the impacts of the mitigating variables in our experiments. The KruskalWallis Test is a non-parametric test that does not require to make assumption about the data set distribution. This test the non-parametric equivalent to the one-way ANOVA and an extension of the Mann-Whitney U Test. It allows the comparison of more than two independent groups.

Chapter 3

Related Work

In this chapter, we summarise previous works on the impact of design smells and their detection techniques. We start with the state-of-the-art of design smells definition, then we present the link between design smells and software maintenance activities which include comprehension and change. Finally, we present the state-of-the-art of design smells detection approaches in perspective with our approach, SMURF.

Contents

3.1 Design Smells Definition and Impact. **21**
3.2 Design Smells Detection. **24**
3.3 Summary **26**

3.1 Design Smells Definition and Impact.

Design Smells Definition. The first book related to design smells in object-oriented development was written in 1995 by Webster [Webster, 1995]; his contribution includes conceptual, political, coding, and quality-assurance problems.

Riel [Riel, 1996] defined 61 heuristics characterising good object-oriented programming to assess a system quality manually and improve its design and implementation.

Brown *et al.* [Brown *et al.*, 1998] described 40 design smells, including the Blob and Spaghetti Code. These books provide in-depth views on heuristics, design smells aimed at industrial and academic audiences.

Beck [Fowler, 1999a] defined 22 code smells, suggesting where developers should apply refactorings. A refactoring is a technique used to improve the internal structure of a program without changing its external behaviour [Fowler, 1999a]. Long method, large class, data class, and long parameter list are examples of code smells. Appendix B provides complete description of code smells and antipatterns.

Mäntylä [Mäntylä, 2003] and Wake [Wake, 2003] proposed classifications for code smells. These works have been important to aware people about the existence of antipatterns and also their potential impact. The description that the authors gave constituted the basis for the proposed detection tools for design smells detection.

Design Smells Impact. Deligiannis *et al.* [2003 ; 2004] proposed the first quantitative study of the impact of antipatterns on software development and maintenance activities. They performed a controlled experiment with 20 students on two systems of the impact of Blob classes on the understandability and maintainability of the systems. The results of their study suggest that Blob classes affect the evolution of design structures and the subjects' use of inheritance. However, Deligiannis *et al.* did not assess the impact of God classes on the ease of their subjects to understand the systems and the subjects' ability to perform successful comprehension tasks on these systems.

Du Bois *et al.* [2006] showed through a controlled experiment with graduate students that the decomposition of God classes into a number of collaborating classes using well-known refactorings can improve their understandability. In their experiment, students were asked to perform simple maintenance tasks on God classes and their decompositions. Du Bois *et al.* found that the students had more difficulties understanding the original God class than certain decompositions. However, their study did not reveal any objective notion of "optimal comprehensibility".

Abbès *et al.* [2011] performed an empirical study to investigate whether the occurrence of design smells does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. They designed and conducted three experiments, with 24 subjects each, to collect data on the performance of developers on basic tasks related to program comprehension and assessed the impact of two antipatterns and of their combinations: Blob and Spaghetti Code. They measured the developers' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data showed that the occurrence of one antipattern does not significantly decrease developers' performance while the combination of two design smells impedes significantly developers' performance. However, they did not study other combinations.

Olbrich *et al.* [2009] analysed the historical data of Lucene and Xerces over several years and concluded that Blob classes and classes subjected to Shotgun Surgery have a higher change frequency than other classes; with Blob classes featuring more changes.

Similarly, Chatzigeorgiou and Manakos [2010] studied the evolution of Long Method, Feature Envy, and State Checking throughout successive versions of two open-source systems and concluded that a significant percentage of these smells are introduced during the addition of new methods to the systems. They also found that these smells persist in systems and that their removal is often a side effect of adaptive maintenance rather than the result of targeted refactoring activities.

Yamashita and Moonen [2012] investigated the extent to which code smells reflect factors affecting software maintainability and observed that using code smell definitions alone, developers cannot fully evaluate the overall maintainability of a software system. They concluded on the need to combine different analysis approaches to achieve more complete and accurate evaluations of the overall maintainability of a software system.

Khomh *et al.* [2012] investigated the impact of design smells on classes in object-oriented systems by studying the relation between the presence of design smells and the change- and fault-proneness of the classes. They detected 13 design smells in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analysed (1) to what extent classes participating in the smells have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the size of the classes or to the presence of smells, and (3) what kinds of changes affect classes participating in smells. They showed that, in almost all releases of the four systems, classes participating in design smells are more change- and fault-prone than others. They also showed that size alone cannot explain the higher odds of classes with design smells to underwent a fault or change compared to other classes.

These previous works show the importance of doing empirical studies on design smells to bring evidences of the negative impact of design smells on software quality attributes and maintenance activities. These works also show that design smells impede maintenance activities from many point of view: design smells are more change and fault prone; then they lead to more corrective maintenance. They impede also the understandability of systems, making change more difficult. Only few studies investigate the impacts of design smells on source code understandability and none of them study their impact on fault-fixing effort. We provide two empirical studies to bring evidences on the impact of design smells on source code understandability and on fault-fixing activities.

3.2 Design Smells Detection.

Several approaches to specify and detect code smells and antipatterns exist in the literature. They range from manual approaches, based on inspection techniques [Travassos *et al.*, 1999], to metric-based heuristics [Marinescu, 2004a ; Munro, 2005 ; Oliveto *et al.*, 2010], using rules and thresholds on various metrics [Moha *et al.*, 2009] or Bayesian belief networks [Foutse Khomh *et al.*, 2009]. Commercial tools such as Borland Together[1] and InCode[2] also enable automatic detections of code smells and design smells.

Detection Techniques. Many researchers studied antipatterns detection. Rahma *et al.* [2011] used quality metrics to identify and predict antipatterns in UML designs using structural and behavioural data.

Ballis *et al.* [2008a ; 2008b] worked on both the detection of design patterns and antipatterns using a rule-based matching approach for extracting all occurrences of a design pattern/antipattern in a graph representation of the source code.

Langelier *et al.* [2005] proposed a visual approach to detect antipatterns. They proposed this approach to deal with complex systems. They claimed that hybrid techniques that combine automatic analysis with human expertise through visualization are excellent alternatives. Based on that, they proposed a visualization framework that supports quality analysis of large-scale software systems.

Marinescu [2004b] presented detection strategies to detect and localise antipatterns in systems. Marinescu defined 10 detection strategies to capture 10 important antipatterns, but suffered from two drawbacks: (1) a user may not successfully detect an antipattern without having extensive knowledge of metric-based rules and (2) different results would be obtained using different thresholds (whose definition itself is difficult).

Dimitrios *et al.* [2012] explored the ways in which the antipattern ontology can be enhanced using Bayesian networks to reinforce the existing ontology-based detection process. Their approach allows software developers to quantify the existence of an antipatterns using Bayesian networks, based on probabilistic knowledge contained in the antipattern ontology regarding relationships of antipatterns through their causes, symptoms and consequences.

Kessentini *et al.* [2010] used search-based techniques to detect antipatterns conjecturing that the more the code deviates from good practices, the more it is likely to be vulnerable to antipatterns, without mentioning the values of recall obtained.

[1] `http://www.borland.com/us/products/together`

[2] `http://www.intooitus.com/products/incode`

Moha *et al.* [2009] proposed an approach based on a set of rules (metrics, relations between classes) that describes the characters of each antipattern to identify them.

Khomh *et al.* [2011b] presented BDTEX (Bayesian Detection Expert), a Goal Question Metric (GQM) based approach to build Bayesian Belief Networks (BBNs) from the definitions of antipatterns. The output of the BBN is a probability that a class is an antipattern or not. For example, the probability of a class being a Blob depends directly on the two symptoms: MainClass and DataClass.

The approaches of Moha *et al.* [2009], Marinescu [2004b], Rahma et *al.* [2011] are mostly based on the use of code/design quality metrics and thresholds to identify antipatterns. The use of these approaches to characterise new antipatterns or to customise existing rules of detection for improving their performance, require practitioners to have an extensive knowledge of design smells, at least about code/design quality metrics and software quality. The knowledge is necessary to derive the rules from the textual description of design smells. Another issue with these approaches is the choice of thresholds whose definition is difficult. Similarly, using the approach proposed by Ballis *et al.* [2008a ; 2008b] requires also to learn the proposed language and to describe structurally and semantically antipatterns using that language. Khomh *et al.* [Khomh *et al.*, 2011b] approach based on Bayesian networks needs extensive knowledge to characterise the antipatterns and built the Bayesian networks. Moreover, based on probability, it cannot tell the user that a class is an antipattern. Then, we will face again the problem of threshold to decide if we consider a class as an occurrence of an antipattern or not based on the probability. The approach of Moha *et al.* [2009] cannot be used successfully on a set of classes because it needs the whole system to automatically set up the thresholds via boxplots. Moreover, none of the above approaches cannot benefit from practitioners' feedback and the only way to improve the performance of one of them is to modify the rules and–or the thresholds of the antipattern formal description.

Support Vector Machines (SVM). SVM has been used in several domains in the past for various applications, *eg.*, bioinformatics [Bedo *et al.*, 2006], information retrieval [Ye *et al.*, 2011], and object recognition [Choi *et al.*, 2012]. It is a recent solution and alternative to the classification problem. For examples, Takashi *et al.* [2006] presented a document retrieval method using non-relevant documents. The approach used an active learning technique based on SVM for evaluating the relevant feedback. Guihong *et al.* [2008] used C-SVM, a variant of SVM, for term classification and proved that expansion terms determined in traditional ways for pseudo-relevance feedback were not all useful. SVM has also been used in image retrieval systems when Sethia *et al.* [2005] used invariant feature histograms to compare the efficiency of different SVMs. They claimed that a significant performance gain was obtained only after several feedback rounds. Kim

et al. [2008] proposed the change classification approach for predicting latent software bugs based on a SVM. Lucia *et al.* [2012] used incremental user feedback to enhance the rate of true positives found while analysing anomaly reports. The approach was validated using the anomaly reports of three real programs and the results showed clear increase in the number of true positives found in the anomaly reports.

Design smells detection is or can be seen as a classification problem where we must distinguish between design smell classes and non-design smell classes. We believe that the success of SVM in resolving classification problems can be exploited to have a better detection for design smells. No previous approaches used SVM to detect design smells.

User Feedback. As with SVM, users' feedback is a mechanism widely used in various domains with good results. In the field of image retrieval, Zhou et al. [2007] used relevance feedback along with SVM to allow users to unblur images. Sethia *et al.* [2005] also used both SVM and relevance feedback to improve image retrieval, exploiting the users' feedback on the initial results to achieve better results based on the feedback. Onoda *et al.* [2006] proposed an approach for an interactive document retrieval using only non-relevant documents information. Basu *et al.* [2008] used a clustering algorithm to iteratively integrate feedback. Similar to SVM, there has been no work on taking into account users feedback in design smells detection.

3.3 Summary

These previous works raised the awareness of the community towards the impact of design smells on software development and maintenance activities. These maintenance activities are very important in a continuous integration development environment, such as agile software development. In this thesis, we build on these previous works and propose experiments assessing the impact of the Blob and Spaghetti Code on the understandability of systems. We then investigate the impact of design smells on the effort required to fix faults in systems. These two studies and evidences will help developers to take rational decisions about dealing with design smells. Based on the impact of design smells and to help developers reduce maintenance costs, several approaches proposed to automatically remove the occurrences of design smells [Moha *et al.*, 2009 ; Marinescu, 2004b ; Rahma Fourati and Abdallah, 2011 ; Khomh *et al.*, 2011b] but these previous approaches have some limitations. Therefore, we finally propose an incremental and iterative approach based on SVM and using users' feedback to accurately detect the occurrences of design smells.

Chapter 4

Evidence of the Impact of Design smells on Comprehension

In the following, we provide details on the study of the impact of design smells on comprehension. We explain the study details and present the results of the empirical study.

Contents

4.1 Context and Problem **28**

4.2 Study Design **30**

4.2.1 Research Questions 30

4.2.2 Objects and Subjects of the Study 31

4.2.3 Independent and Mitigating Variables 33

4.2.4 Dependent Variables 34

4.2.5 Study Design and Procedure 34

4.2.6 Analysis Method 37

4.3 Study Results **37**

4.3.1 Impact of Two Occurrences of the Blob on Comprehension 37

4.3.2 Impact of Two Occurrences of the Spaghetti code on Comprehension 38

4.4 Discussion **39**

4.4.1 Interpretation of the Results 40

4.4.2 Impact of the Mitigating Variables 41

4.5 Threats to Validity **42**

4.5.1 Construct Validity 42

4.5.2 Internal Validity 42

4.5.3 Conclusion Validity . . . 43
4.5.4 Reliability Validity . . . 43
4.5.5 External Validity . . . 43
4.6 Summary . . . **44**

4.1 Context and Problem

In theory, design smells are "poor" solutions to recurring design problems; they stem from experienced software developers' expertise and describe common pitfalls in object-oriented programming, *eg.*, Brown's 40 design smells [Brown *et al.*, 1998]. Design smells are generally introduced in software systems by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Coplien [Coplien and Harrison, 2005] described a design smell as "something that looks like a good idea, but which back-fires badly when applied".

Examples of Design smells: An example of design smell is the Blob, also called God Class. The Blob is a large and complex class that centralises the behavior of a portion of a software system and only uses other classes as data holders, *i.e.*, data classes. The main characteristics of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and–or accessors to their fields. Another example of design smell is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. Spaghetti Code classes do not exploit, and in some cases prevent the use of object-orientation mechanisms, *i.e.*, polymorphism and inheritance.

Premise: Design smells are conjectured in the literature to decrease the quality of software systems. Yet, despite the many studies on design smells, few studies have empirically investigated the impact of design smells on program comprehension. Program comprehension is central to an effective software maintenance and evolution [von Mayrhauser and Vans, 1995], specially in the context of agile software development which is an iterative and incremental development environment. Indeed, a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension. Therefore, a better understanding of the factors affecting developers's comprehension of source code is an efficient and effective way to ease maintenance. In an effort to address this gap in prior literature, we conducted two experiments with 59 subjects. We analysed whether increasing the number of occurrences of same design smells, in

particular two occurrences of the Blob or two occurrences of the Spaghetti Codes, in the systems impact comprehension.

Goal: Comprehension is an important activity for software maintenance and evolution. Comprehension of a system is the degree to which its source code can easily be understood by developers [Khomh and Guéhéneuc, 2008]. We want to gather quantitative evidence of the impact of Blob and Spaghetti Code design smells on comprehension. Gathering evidence of the relation between Blob and Spaghetti Code design smells and comprehension is important for practitioners, *i.e.*, developers, testers, and managers and will contribute to reduce software development costs and improve software quality. Indeed, the knowledge of the impact of design smells on comprehension will help practitioners to take rational decisions about how dealing with design smells. Thus, evidence of a negative impact of design smells on comprehension, will allow us to advise practitioners to be more cautions with design smells, *i.e.*, with their design decisions (or lack thereof).

Study: We conduct two experiments: we study whether systems with two Blobs, first, and two Spaghetti Codes, second, are more difficult to understand than systems without any design smell. The experiments are performed with 59 subjects and on six different systems developed in Java. The subjects are graduate students and professional developers with experience in software development and maintenance. We ask the subjects to perform three different program comprehension tasks covering three out of four categories of usual comprehension questions [Sillito, 2007]. We measure the subjects' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers.

Results: Collected data of this study show that increasing the number of occurrences of a Blob or a Spaghetti Code design smell impacts comprehension.

Relevance: Understanding the impact of Blob and Spaghetti Code design smells on comprehension is important from the points of view of both researchers and practitioners. For researchers, our results bring further evidence to support the conjecture in the literature on the negative impact of design smells on comprehension. For practitioners, our results provide concrete evidence that they should pay attention to systems with a high number of classes participating in design smells, because these design smells would reduce their comprehension and, consequently, increase their systems' aging [Parnas, 1994]. Our results also support *a posteriori* the removal of design smells as early as possible from systems and, therefore, the importance and usefulness of design smells detection techniques.

4.2 Study Design

Abbès *et al.* [2011] showed that co-occurrence of Blob and Spaghetti Code, impacts program understandability. However, it is not clear whether the reported negative impact of design smells on system understandability is due to the density of design smells in the systems and – or to the occurrences of the specific design smells together. Thus, we cannot claim, from the study conducted by Abbès *et al.* [2011], that increasing same type of design smells could impact system understandability.

In this study, we further examine the impact of occurrences of the same design smell (Blob or Spaghetti Code) on comprehension. Precisely, we replicate the experiment done by Abbès *et al.* and conduct two experiments to analyse whether two occurrences of Blobs or two occurrences Spaghetti Code, in a single system, impacts comprehension. This study will provide confidence and more detailed view on the study conducted by Abbès *et al.* [2011]. We perform two experiments: Experiment 1 deals with two occurrences of Blob and Experiment 2 deals with two occurrences of Spaghetti Code in a system. We choose these design smells because they are well-known and have been used in the literature to perform other experiments, in particular in the previous work by [Khomh *et al.*, 2009a ; Bois *et al.*, 2006 ; Moha and Guéhéneuc, 2005 ; Vaucher *et al.*, 2009 ; Moha *et al.*, 2010a ; Abbes *et al.*, 2011]. In each experiment, we assign two systems to each subject: one containing two occurrences of Blob or Spaghetti Code design smell and one without any design smell. We then measure and compare the subjects' performances for both systems. We follow Wohlin *et al.*'s template [2000] to describe the experimental design.

4.2.1 Research Questions

Our research questions stems from our goal of understanding the impact of two occurrences of the same design smell on on program comprehension. We choose the Blob and Spaghetti Code design smells because we conducted a replication of the experiment of Abbès *et al.* and also to compare the results with previous studies done with these design smells. Therefore, we can declined our research question in two research questions:

- **RQ1** - what is the impact of two occurrences of the Blob design smell on comprehension?
- **RQ2** - what is the impact of two occurrences of the Spaghetti Code design smell on comprehension?

To answer our research questions, we perform two experiments with a total 59 subjects. Each subject is given two systems, one with and one without Blob and Spaghetti Code design smells, to

perform program comprehension tasks. For RQ1, we use Blob design smell and for RQ2 we use Spaghetti Code design smell. We record each subject's total correct answers, total time, and total effort to perform program comprehension tasks for each system.

When designing our experiment, our first reflection was to formulate our hypothesis properly as this guide us to a good design. According to our goal, we want to assess null hypothesis when subjects perform comprehension tasks with source code. We perform statistical tests to accept or reject the null hypothesis. For RQ1, we formulate the following null hypotheses:

- $H01_{2Blobs}$: There is no statistically significant difference between the subjects' average correct answers when executing comprehension tasks on the source code of systems containing two occurrences of Blob design smell compared to systems without any design smells.
- $H02_{2Blobs}$: There is no statistically significant difference between the subjects' average time spent when executing comprehension tasks on the source code of systems containing two occurrences of Blob design smell compared to systems without any design smells.
- $H03_{2Blobs}$: There is no statistically significant difference between the subjects' average effort spent when executing comprehension tasks on the source code of systems containing two occurrences of Blob design smell compared to systems without any design smells.

For RQ2, we formulate the following null hypotheses:

- $H01_{2Spaghettis}$: There is no statistically significant difference between the subjects' average correct answers when executing comprehension tasks on the source code of systems containing two occurrences of Spaghetti design smell compared to systems without any design smells.
- $H02_{2Spaghettis}$: There is no statistically significant difference between the subjects' average time spent when executing comprehension tasks on the source code of systems containing two occurrences of Spaghetti design smell compared to systems without any design smells.
- $H03_{2Spaghettis}$: There is no statistically significant difference between the subjects' average effort spent when executing comprehension tasks on the source code of systems containing two occurrences of Spaghetti design smell compared to systems without any design smells.

4.2.2 Objects and Subjects of the Study

Objects of the Study. We choose three systems for each experiment, all developed in Java, and briefly described in Table 4.1. We performed each experiment on 3 systems, because one system could be intrinsically easier/more complex to understand.

Experiments	Systems	Classes	SLOCs	Release
1	Azureus v2.3.0.6	1,449	191,963	2005
	iTrust v11.0	565	21,901	2010
	SIP v1.0	1,771	486,966	2010
2	ArgoUml v0.20	1,230	113,017	2006
	JHotDraw v5.4b2	484	72,312	2004
	Rhino v1.6R5	108	48,824	2009

Table 4.1 – Object Systems.

For Experiment 1, we use Azureus[1]: (now known as Vuze) a Bit Torrent client used to transfer files via the Bit Torrent protocol; iTrust[2]: a medical application that provides patients with a means to keep up with their medical history and records as well as communicate with their doctors, including selecting which doctors to be their primary care-giver, seeing and sharing satisfaction results, and other tasks; and, SIP[3]: (now known as jitsi) an audio/video Internet phone and instant messenger that supports some of the most popular instant messaging and telephony protocols.

For Experiment 2, we use ArgoUML[4]: a UML diagramming application written in Java; JHotDraw[5]: a graphic framework for drawing 2D graphics; and, Rhino[6]: an open-source implementation of a JavaScript interpreter.

We used the following criteria to select the systems. First, we selected open-source systems; therefore other researchers can replicate our experiment. Second, we avoided to select small systems that do not represent the ones that developers deal with normally. We randomly assigned a set of three systems to each experiment. We also chose these systems because they are typical examples of systems having continuously evolved on periods of time of different lengths. Hence, the occurrences of Blob and Spaghetti Code in these systems are not coincidence but are realistic. We use the design smell detection technique, DETEX, which stems from the DECOR method [Moha and Guéhéneuc, 2005 ; Moha *et al.*, 2010a] to ensure that each system has two occurrences of the Blob and–or the Spaghetti Code design smell. We manually validated the detected occurrences and we manually verified that the remaining classes are not occurrences of design smell. From each system, we randomly selected a subset of classes responsible for managing a specific task to limit the size of the displayed source code to the subjects. For example, in iTrust we chose the source code of the classes responsible for providing patients with a means to keep up with their medical history and records as well as communicate with their doctors.

[1] `http://www.vuze.com/`
[2] `http://agile.csc.ncsu.edu/iTrust/`
[3] `http://www.jitsi.org/`
[4] `http://argouml.tigris.org/`
[5] `http://www.jhotdraw.org/`
[6] `http://www.mozilla.org/rhino/`

However this difference would not impact our results because, regardless of the sizes, a subject concentrates his efforts only on a small part of the subset in which a Blob or Spaghetti Code class plays a *central* role, *i.e.*, the Blob class and its surrounding classes. Therefore, we ensure that all subjects perform the comprehension tasks within, almost, the same piece of code. Then, we refactor [Fowler, 1999b] each subset of each system to remove all other occurrences of (other) design smells to reduce possible bias by other design smells, while keeping the system compilable and functional. We performed manual refactoring following the guidelines of Fowler's book [Fowler, 1999b]. For example when dealing with a Blob class, we replace it by multiple smaller classes or just spread the methods to their right places.

Therefore, for Experiment 1, each subset contains two occurrences of the Blob. For Experiment 2, each subset contains two occurrences of the Spaghetti Code. We finally refactor each subset of the systems to obtain new subsets in which no occurrence of the design smells exist. We use these subsets as base line to compare the subjects' performance and test our null hypothesis.

Subjects of the Study. Both experiments were performed by 59 anonymous subjects. Some subjects were enrolled in the M.Sc. and Ph.D. programs in the Computer and Software Engineering Department in École Polytechnique de Montréal, in the Computer Science Department in Université de Montréal, and the School of Computer Science in Carleton University. Others were professionals working for software companies in the Montréal area. All subjects were volunteers and could withdraw at any time, for any reason.

4.2.3 Independent and Mitigating Variables

The independent variable in Experiment 1 is the presence or not of two occurrences of a Blob design smell, which is a Boolean value stating whether the source code contain two occurrences of Blob or not. It is the value of this independent variable that should influence the subjects' performances. In Experiment 2, the independent variable is the presence or not of two occurrences of a Spaghetti Code design smell in the source code.

We retain three mitigating variables possibly impacting the measures of the dependent variables:

- Subject's knowledge level in Java.
- Subject's knowledge level of Eclipse.
- Subject's knowledge level in software engineering.

	With Design smells	Without Design smells
System 1	$S_3, S_7, S_9, S_{11}, S_{12}, S_{18}, S_{21}, S_{24}$	$S_1, S_5, S_8, S_{10}, S_{15}, S_{16}, S_{20}, S_{22}$
System 2	$S_1, S_2, S_6, S_{14}, S_{15}, S_{17}, S_{20}, S_{22}$	$S_4, S_7, S_9, S_{11}, S_{13}, S_{18}, S_{19}, S_{23}$
System 3	$S_4, S_5, S_8, S_{10}, S_{13}, S_{16}, S_{19}, S_{23}$	$S_2, S_3, S_6, S_{12}, S_{14}, S_{17}, S_{21}, S_{24}$

Table 4.2 – Experimental design.

We assess the subjects' levels using a *post-mortem* questionnaire administered to subjects at the end of their participation to our study to avoid any bias, because some questions pertain to design smells. This questionnaire uses Likert scales for each of the mitigating variables and also include open questions about design smells, refactorings, and so on.

4.2.4 Dependent Variables

The dependent variables measure the subjects' performance, in terms of effort, time spent, and percentage of correct answers.

We measure the subjects' effort using the NASA Task Load Index (TLX) [Hart and Stavenland, 1988]. The TLX assesses the subjective workload of subjects. It is a multi-dimensional measure that provides an overall workload index based on a weighted average of ratings on six sub-scales: mental demands, physical demands, temporal demands, own performance, effort, and frustration. The details on these sub-scales are explained in Appendix D. We combine weights and ratings provided by the subjects into an overall weighted workload index by multiplying ratings and weights; the sum of the weighted ratings divided by fifteen (sum of the weights) represents the effort [Hart and Stavenland, 1988].

We measure the time using a timer developed in Java that the subjects must start before performing their comprehension tasks to answer the questions and stop when done.

We compute the percentage of correct answers for each question by dividing the number of correct elements found by a subject by the total number of correct elements they should have found. For example, for a question on the references to a given object, if there are ten references but the subject found only four, the percentage would be forty.

4.2.5 Study Design and Procedure

Study Design. Our design is a 2×3 factorial design [Wohlin *et al.*, 2000], presented in Table 4.2. In 2×3 factorial design there cannot be more combinations than detailed in Table 4.2. Thus, we randomly select any combination from S1 to S24 for the subjects from S25 to S30. We have

three different systems, each with two possibilities: containing or not containing the occurrences of the design smell. Hence, six combinations are possible. For each combination, we prepare a set of comprehension questions, which together form a *treatment*. We have six different groups of subjects, each one affected by each one treatment.

This design is a between-subject design [Duchowski, 2007] with a set of different groups of subjects, which allows us to avoid repetition by using a different group of subjects for each treatment. We take care of the groups to ensure their homogeneity and avoid bias in the results, for example we ensure that no group entirely contains male or female subjects. The use of balanced groups simplifies and enhances the statistical analysis of the collected data [Wohlin *et al.*, 2000].

Procedure. We received the agreement from the Ethical Review Boards of Université de Montréal and Carleton University to perform and publish this study. The collected data is anonymous. The subjects could leave any experiment at any time, for any reason, and without penalty of any kind. No subject left the study or took more than 45 minutes to perform the experiment. The subjects knew that they would perform comprehension tasks, but did know neither the goal of the experiment nor whether the system that they were studying contained or did not contain design smells. We informed them of the goal of the study after collecting their data, before they finished the experiment.

For each experiment, we prepare an Eclipse workspace packaging the target classes, on which the subjects must perform their comprehension tasks to answer the selected questions. The workspace contains compilable and functional subsets, linked to JAR files with the rest of the system compiled code. It also includes the timer, the TLX program, a brief tutorial on the use of Eclipse, a brief explanation about the system at hand, and the post-mortem questionnaire. We conduct the experiments in the same lab, with the same computers and software environments to avoid any kind of environmental bias. No subjects knew the systems on which they perform comprehension tasks, thus we eliminate the mitigating variable relative to the subject's knowledge of the system.

Questions. We used comprehension questions to elicit comprehension tasks and collect data on the subjects' performances.

We consider questions in three of the four categories of questions regularly asked and answered by developers [Sillito, 2007]: (1) finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task; (2) focusing on a particular class believed to be related to some task and on directly-related classes; (3) understanding a number of classes and their relations in some subset of the source code; and, (4) understanding the relations between

different subsets of the source code. Each category contains several questions of the same type [Sillito, 2007].

We choose questions only in the first three categories, because the last category pertains to different subsets of the source code and, in our experiments, we focus only on one subset containing the occurrence(s) of the design smell(s). In each chosen category, we select the two most relevant questions through votes among four Ph.D student in software engineering. Selecting two questions in each category allows us to have, for each subject, a different question from the same category on the system with and without design smell, hence reducing the possibility of a learning bias for the second system.

The six questions are the followings. The text in bold is a placeholder that we replace by appropriate behaviors, concepts, elements, methods, and types depending on the systems on which the subjects were performing their tasks.

- Category 1: Finding focus points:
  - Question 1: Where is the code involved in the implementation of **this behavior**?
  - Question 2: Which type represents **this domain** concept or **this UI element or action**?
- Category 2: Expanding focus points:
  - Question 1: Where is **this method** called or **this type** referenced?
  - Question 2: What data can we access from **this object**?
- Category 3: Understanding a subset:
  - Question 1: How are **these types or objects** related?
  - Question 2: What is the behavior that **these types** provide together and how is it distributed over **these types**?

For example, with Azureus, we replace "**this behavior**" in Question 1, Category 1, by "indicating the health status of the resource to be downloaded, by calculating the number of seeds and peers" and the question reads: "Where is, in this project, the method involved in the implementation of indicating the health status of the resource to be downloaded, by calculating the number of seeds and peers?"

For category 2, it may seem that the questions could be simply answered by subjects using Eclipse. Yet, subjects still must identify and understand the classes or methods that they believe to be related to the task. Moreover, discovering classes and relationships that capture incoming connections prepare the subject for the questions of the third category. We provide in Appendix C, example of questions for Azureus.

4.2.6 Analysis Method

We use the (non-parametric) Mann-Whitney test to compare two sets of dependent variables and assess whether their difference is statistically significant, see Section 2.5.2. The two sets are the subjects' data collected when they answer the comprehension questions on the system with two occurrences of Blob or Spaghetti Code design smell and without. For example, we compute the Mann-Whitney test to compare the set of times measured for each subject on the system with with two occurrences of Blob design smell with the set of times measured for each subject on the system without any design smell. Other than testing the hypotheses, we also estimate the magnitude of the difference in means between the two groups. We use the non-parametric effect size measure Cliff's d, which indicates the magnitude of the effect size of the treatment on the dependent variable, see Section 2.5.4.

We use KruskalWallis one-way analysis of variance by ranks to analyse the impacts of the mitigating variables, see Section 2.5.5.

4.3 Study Results

We now present the results of the study, answer each research question, and try to reject the null hypothesis. Our motivation for conducting this study was to confirm and explain if the impact of the combination of Blob and Spaghetti Code design smells observed in the Abbès *et al.*'s study [2011] was due to the number of occurrences or the number of different kind of design smells. We also want to discuss the impact of the increase of the number of occurrences of the same design smells compared to the result obtained in the Abbès *et al.*'s study.

4.3.1 Impact of Two Occurrences of the Blob on Comprehension

Table 4.3 – Mann-Whitney tests and Cliff's d results for Experiment 1.

	M.-W. p	t-Test p	Cliff d
Times	$<$**0.01**	$<$**0.01**	**0.87**
Answers	$<$**0.01**	$<$**0.01**	**0.53**
Efforts	$<$**0.01**	$<$**0.01**	**0.73**

For experiment 1, Table 4.3 reports the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the three indicators of performance (effort, time spent, and percentage of correct

answers) of the subjects working on the systems with and without two occurrences of Blob design smells.

The results show that :

- there is a statistically significant difference between **the average effort** of the subjects working on a system with two occurrences of Blob and that of the subjects working on a system without any occurrences of design smells with large effect size
- there is a statistically significant difference between **the average time spent** by the subjects working on a system with two occurrences of Blob and that of the subjects working on a system without any occurrences of design smells with large effect size
- there is a statistically significant difference between **the average percentage of correct answers** of the subjects working on a system with two occurrences of Blob and that of the subjects working on a system without any occurrences of design smells with large effect size.

Figure 4.1 illustrates the statistical differences of the dependent variables: (1) the subjects' efforts are higher (2) the times spent by the subjects to perform the comprehension tasks are higher; and, (3) the percentages of correct answers are lower, in the system with the two occurrences of Blob design smells. Moreover, Cliff *d* effect size values are large (**0.71** in average).

We thus can reject the null hypotheses $H01_{2Blobs}$,$H02_{2Blobs}$, $H03_{2Blobs}$ and answer our RQ1 that the subjects working with a system with two occurrences of blob put more effort and spend more time but have less percentage of correct answers than the subjects working with a system without any design smell. The two occurrences of the Blob design smells have a strong impact on subjects' efforts, times, and percentages of correct answers. A possible explanation is that *two occurrences of the Blob design smells impedes the subjects' comprehension*

4.3.2 Impact of Two Occurrences of the Spaghetti code on Comprehension

Table 4.4 – Mann-Whitney tests and Cliff's *d* results for Experiment 2.

	M.-W. p	t-Test p	Cliff d
Times	**<0.01**	**<0.01**	**0.93**
Answers	**<0.01**	**<0.01**	**0.93**
Efforts	**<0.01**	**<0.01**	**0.55**

For experiment 2, Table 4.4 reports the results of the Mann-Whitney tests and Cliff's *d* effect sizes to compare the three indicators of performance (effort, time spent, and percentage of correct answers) of the subjects working on the systems with and without two occurrences of Spaghetti Code design smells.

For experiment 2, the results show that :

- there is a statistically significant difference between **the average effort** of the subjects working on a system with two occurrences of Spaghetti Code and that of the subjects working on a system without any occurrences of design smells with large effect size
- there is a statistically significant difference between **the average time spent** of the subjects working on a system with two occurrences of spaghetti Code and and that of the subjects working on a system without any occurrences of design smells with large effect size
- there is a statistically significant difference between **the average percentage of correct answers** of the subjects working on a system with two occurrences of spaghetti Code and and that of the subjects working on a system without any occurrences of design smells with large effect size.

Figure 4.1 illustrates the statistical differences of the dependent variables: (1) the subjects' efforts are higher (2) the times spent by the subjects to perform the comprehension tasks are higher; and, (3) the percentages of correct answers are lower, in the system with the two occurrences of Spaghetti Code design smells. Moreover, Cliff *d* effect size values are large (**0.80** in average).

We thus can reject the null hypotheses $H01_{2Spaghettis}$,$H02_{2Spaghettis}$, $H03_{2Spaghettis}$ and answer our RQ2 that the subjects working with a system with two occurrences of Spaghetti Code put more effort and spend more time but have less percentage of correct answers than the subjects working with a system without any design smell. The two occurrences of the Spaghetti Code design smells have a strong impact on subjects' efforts, times, and percentages of correct answers. A possible explanation is that *two occurrences of the Spaghetti Code design smells impedes the subjects' comprehension*

4.4 Discussion

This section discusses the results reported in Section 4.3, summarised in Table 4.5. We structure the discussion as follows: first, we discuss the interpretation of our results compared to the results of Abbès *et al.* study, and second, we discuss the impact of the mitigating variables in our experiments and results.

Table 4.5 – Summary of the statistically-significant results.

	Answers	Effort	Time
RQ1	✓	✓	✓
RQ2	✓	✓	✓

4.4.1 Interpretation of the Results

Abbès *et al.*'s study [2011] concluded that one occurrence of Blob or Spaghetti Code doesn't significantly affect the subjects' comprehension of the source code. We found in our study that when we increase the number of occurrences of the design smells (Blob or Spaghetti Code), we observe significant differences in subjects' efforts, times, and percentages of correct answers with very large effect size. This finding in our study confirm the fact that the more occurrences of the Blob and Spaghetti Code design smells we have in the source code, the higher the complexity of source code and therefore, the comprehension of the subjects will be negatively impacted. For example some design smells directly impact the implementation of a method/class, such as Spaghetti Code, which is a symptom of complex logics with interlocked control flows. Thus, the occurrences of such design smells could directly lead to more complexity and, thus, impact negatively program comprehension. Also, the Blob design smell is characterized by a large class (God class) which is not cohesive enough. This class centralizes most of the processing, takes most of the decisions, and is associated to data classes. Thus, when developers must fix a bug in such a class, they must check all the "surrounding" classes to make sure that they understand all the different links so that when changing this class, they can adapt all these "surrounding" classes to the changes. These characteristics could lead to difficulty to understand the code source and thus, could make developers spending more effort to understand them when they need to fix them and spend more time without finding sometimes the correct information.

Abbès *et al.*'s study [2011] concluded that a combination of the Blob and the Spaghetti Code design smells has a strong impact on subjects' efforts, times, and percentages of correct answers. With our study, we found that two occurrences of same design smell (Blob or Spaghetti Code) has also a strong impact on subjects' efforts, times, and percentages of correct answers. This findings explicit the results obtained in Abbès *et al.*'s study and rather extend it.

We can say that two occurrences of Blob or Spaghetti Code design smell in code source is sufficient to negatively impact the maintainers' efforts, times, and percentages of correct answers no matter if these two occurrences are from the same design smell or from two different kind of design smell.

	Systems	Efforts	Time	% of Correct Answers
Java Knowledge	With Blob	0.08	0.06	0.12
	Without Blob	0.78	0.27	0.42
	With Spaghetti	0.24	0.75	0.41
	Without Spaghetti	0.70	0.67	0.30
Eclipse Knowledge	With Blob	0.19	0.44	0.46
	Without Blob	0.54	0.33	0.28
	With Spaghetti	0.18	0.21	0.26
	Without Spaghetti	0.86	0.21	0.31
Software Engineering Knowledge	With Blob	0.37	0.32	0.43
	Without Blob	0.47	0.15	0.05
	With Spaghetti	0.33	0.48	0.55
	Without Spaghetti	0.88	0.76	0.33

Table 4.6 – Kruskal-Wallis *p*-values of the impact of knowledge levels (AP = Design smell).

We used statistically-significant results of RQ1, and RQ2 to warn developers about the negative impact on programm comprehension of co-occurrence of two Blob or two Spaghetti Code design smells. Developers should thus be wary of and track these design smells. Because the number of occurrences of design smells is related to difficulty to understand the source code. Thus, correcting these design smells may help in understanding the source code, reducing the time spent for a task and the effort and increasing their ability to find the correct information.

4.4.2 Impact of the Mitigating Variables

We investigated if the three mitigating variables: Java knowledge, Eclipse knowledge, and software engineering knowledge, impacted our results. We set 5 levels, using Lickert scales, corresponding to the subjects' respective levels (bad, neutral, good, excellent, expert).

We performed Kruskal-Wallis one-way analysis of variance by ranks test to assess the impact of the mitigating variables on the three measured variables (time, effort, and % of correct answers), which shows that the mitigating variables do not impact our results, as shown by the high *p*-values in Table 4.6. The lack of impact of these mitigating variables is consistent with previous findings [Cepeda Porras and Guéhéneuc, 2010], in which the authors assessed the impact of some subjects' knowledge on design patterns on the understandability of various representations of software systems.

4.5 Threats to Validity

Some threats limit the validity of our study. We now discuss these threats and how we alleviate or accept them following common guidelines provided in [Wohlin *et al.*, 2000].

4.5.1 Construct Validity

Construct validity threats concern the relation between theory and observations. In this study, they could be due to measurement errors. We use times and percentages of correct answer to measure the subjects' performances. These measures are objective, even if small variations due to external factors, such as fatigue, could impact their values. We also use the TLX to measure the subjects' effort. The TLX is by its very nature subjective and, thus, it is possible that our subjects provided us with particular effort values.

The degree of seriousness of the design smells is also a threat to construct validity. The Blob and Spaghetti Code, in each system, were validated through a voting process for decisions. Four Ph.D. students voted for the design smells. We follow the definitions provided in the book of Brown *et al.* [1998] to deal with design smells. Yet the occurrences used could have been more or less serious than in other systems. Future work should mitigate this threat.

Construct validity threats could also be due to a mistaken relation between design smells and system understandability. We believe that this threat is mitigated by the facts that many authors discussed this relation, that this relation seems rational, and that the results of our analysis tend to show that, indeed, design smells impact system understandability.

4.5.2 Internal Validity

We identify four threats to the internal validity of our study: learning, selection, instrumentation, and diffusion.

Learning threats do not affect our study for a specific experiment because we used a between-subject design. A between-subject design uses different groups of subjects, to whom different treatments are assigned. We also took care to randomize the subjects to avoid bias (*eg.*, gender bias). Each subject performed comprehension tasks on two different systems with different questions for each system. However, the same subjects performed Experiment 1 and Experiment 2. The learning effect is minimal because in the Experiment 2 we used different systems and different questions.

Selection threats could impact our study due to the natural difference among the subjects' abilities. We tried to mitigate this threat by asking only volunteers, therefore with a clear willingness to participate. We also studied the possible impact of their levels of knowledge in Java, of Eclipse, and in Software engineering, through three mitigating variables without obtaining any statistically significant results.

Instrumentation threats were minimized by using objective measures like times and percentages of correct answers. We observed some subjectivity in measuring the subjects' effort via TLX because, for instance, one subject 100% effort could correspond to another's 50% of effort. However, this subjectivity illustrates the concrete feeling of effort of the subjects.

Diffusion threats do not impact our study because we asked subjects not to talk about the study among themselves and the systems and questions among experiments were different. However, all the subjects attend the experiment at the same moment so it is impossible that the subjects exchanged some information.

4.5.3 Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. Also, we mainly used non-parametric tests that do not require to make assumption about the data set distribution.

4.5.4 Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. The systems, questionnaires, and raw data to compute the statistics are on-line[7].

4.5.5 External Validity

We performed our study on six different real systems belonging to different domains and with different sizes, see Table 4.1. Our design, *i.e.*, providing only on average 75 classes of each system to each subject, is reasonable because, in real maintenance projects, developers perform their tasks on small parts of whole systems and probably would limit themselves as much as possible to avoid getting "lost" in large code base. However, we cannot assert that our results can be generalised to other Java systems, systems in other programming languages, and to other subjects; future work

[7] `http://www.ptidej.net/downloads/experiments/emse12r/`

includes replicating this study in other contexts, with other subjects, other questions, other design smells, and other systems.

4.6 Summary

Design smells are conjectured in the literature to negatively impact the quality of systems. Few previous studies have empirically investigated the impact of design smells on program comprehension. We revisit studies on the impact of Blob and Spaghetti Code design smells on program comprehension. We design and conduct two experiments with 59 subjects, to assess the impact of two occurrences Blob or Spaghetti Code design smells, on the developers' performance during program comprehension tasks. We measure developers' performance using: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. The results of the two experiments confirm and explicit previous findings by Abbès *et al.*, that the combination of two Blob or two Spaghetti Code design smells impedes significantly developers performance during comprehension and maintenance tasks. The obtained results justify a posteriori previous researches on the specification and detection of design smells. Software development teams should warn developers against increasing the number of design smells in software systems and recommend refactorings to remove them when possible. Consequently, developers and quality assurance personnel should be wary with growing numbers of Blob and Spaghetti Code design smells in their systems as they could reduce their system understandability and, therefore, increase the risks of the systems aging and also the introduction of faults. Indeed, D'Ambros *et al.* [D'Ambros *et al.*, 2010] found that an increase in the number of design smells in a system is likely to generate faults.

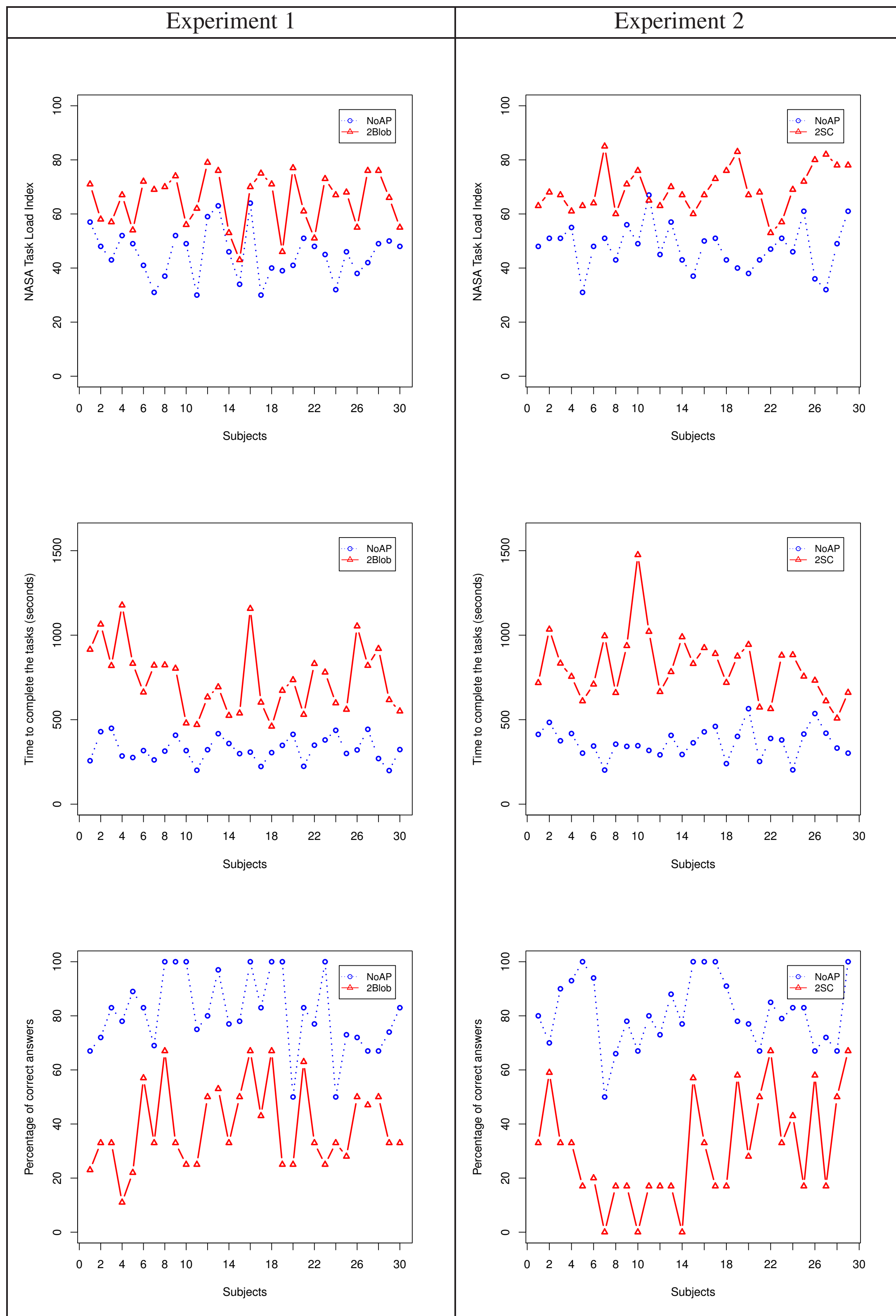


Figure 4.1 – Graphical representations of the collected data.

Chapter 5

Evidence of the Impact of Design Smells on Fault-fixing Efforts

In the following, we provide details on the study of the impact of design smells on fault-fixing activities. We explain the study details and present the results of the empirical study.

Contents

5.1 Context and Problem **47**
5.2 Study Design **49**
5.2.1 Research Questions 49
5.2.2 Independent Variables 52
5.2.3 Dependent Variables 53
5.2.4 Analysis Method 55
5.3 Study Results **57**
5.3.1 Relation between Design Smells and Faults 58
5.3.2 Relation between Design Smells and the Duration of Fixing Periods . 60
5.3.3 Relation between Design Smells and the Number of Impacted Elements 61
5.3.4 Relation between Design Smells and Fault-fixing Entropy 62
5.3.5 Relation between Fault-fixing and the Number of Occurrences of Design Smells 64
5.4 Discussion **65**
5.4.1 Statistically-significant Results 65
5.4.2 Non-statistically-significant Results 68
5.4.3 General Discussions 69

5.5 Threats to validity **69**
5.5.1 Construct validity 69
5.5.2 Internal Validity 70
5.5.3 Conclusion Validity 70
5.5.4 Reliability Validity 70
5.5.5 External Validity 70
5.6 Summary **70**

5.1 Context and Problem

Context. Design smells are poor solutions to recurring design or implementation problems. Design smells reflect developers' poor expertise. They are generally the result of misuse of the object-oriented paradigm and–or design patterns. An example of design smell is the *Blob*, also called *God Class*, which is a large and complex class that centralises most of the behavior of a system. A Blob also depends on data stored in surrounding *data classes*. A Blob is characterized by a low cohesion and a large size. It prevents the use of polymorphism through inheritance [Moha *et al.*, 2010c]. Another example of design smell is the *MessageChain*, which occurs when the implementation of a functionality in one class requires a long chain of method invocations between objects from different classes [Moha *et al.*, 2010c]. Classes with message chains are more prone to faults [Khomh *et al.*, 2011a].

Background. Design smells are prevalent in systems [Khomh *et al.*, 2011a]. Several studies [Abbes *et al.*, 2011 ; Khomh *et al.*, 2011a] have shown that design smells negatively impact software development and maintenance. Khomh *et al.* [Khomh *et al.*, 2011a] showed that classes participating in design smells are more prone to changes and faults than others. Many (other) studies on design smells and software fault exist and are summarised in chapter 3. However, none of them empirically investigated the impact of design smells on the developers' effort to fix faults.

Premise and Goal. Following previous work on the negative impact of design smells on class change and fault-proneness and because software maintenance is the most expensive part of any software development effort [PressmanR.S., 1996], we suppose the following premise:

> There is a relation between design smells, faults, and developers' effort to fix faults.

Validating this premise is important for practitioners, *i.e.*, developers, testers, and managers. Indeed, if we bring evidences on the negative impact of design smells on fault-fixing activities and

on a relation between the fixing of faults and the number of occurrences of design smells, then we could advise practitioners to be more cautions with design smells, *i.e.*, with their design decisions (or lack thereof). These evidences could help practitioners justify the removal of design smells and would also further confirm, a posteriori, previous statements on the negative impact of design smells on software development and maintenance, and highlight the importance of design smell detection.

We want to gather quantitative evidence on the relation between design smells, faults, and the developers' effort to fix the faults.

Study. First, we conduct a preliminary study on the relation between design smells and faults. This study replicates a previous study [Khomh *et al.*, 2011a] and forms the ground on which we build the rest of our study. It confirms the relation between the number of occurrences of design smells in a class and its fault-proneness. This study allows us to gather the occurrences of design smells needed for the rest of the study and gives us confidence in this collected occurrences because we can reproduce previous results. Second, we compute data related to the developers' effort to fix faults: the durations of the fixing periods of faults, the number of fields and methods impacted by the fault fixes, and the entropies of the fault fixes. The entropy of a fault fix captures the distribution of the code modified during the fix across the files of a system. Third, we relate occurrences of design smells—their numbers and kinds—with our measures of the developers' effort to fix faults using (non-parametric) Mann-Whitney tests as well as multivariate regression analyses. We perform our study on 10 releases of ArgoUML, 13 of Eclipse, 18 of Mylyn, and 13 of Rhino. In total, we analyze 7,797 faults and their fixes and 106,776 occurrences of 12 different design smells in the 26,793 different classes of the studied systems.

Results and Relevance. The results of our analyses lead us to conclude that, generally, design smells increase the developers' effort to fix faults and, thus, to confirm our premise. This increase in effort is due to increases in the duration of fault fixes and in the complexity of the fixes. Understanding the impact of design smells on developers' fault-fixing efforts is important for both researchers and practitioners. For researchers, our study brings evidence that design smells are indeed *negative* for software development and maintenance. Our results confirm previous observations that there is a higher number of occurrences of design smells in fault-prone classes than in non fault-prone classes. We also bring concrete, novel evidence of the impact of design smells on developers' fault-fixing effort and further justify a posteriori previous work on the detection of design smells. For practitioners, we provide concrete, novel evidence that they should pay attention to the presence of design smells in their systems and try to avoid them or, at least, correct them before fixing faults. Design smells increase developers' fault-fixing effort and, consequently,

may increase software costs. Managers could use design-smell detection techniques to assess the volume of classes participating in design smells in to-be-acquired systems and, thus, adjust their offers and forecast the systems cost-of-ownerships and–or plan for refactoring because the design smells will impact fault-fixing effort.

5.2 Study Design

We formulate five research questions (RQs) and ten hypotheses related to the relations between the presence of occurrences of design smells and the developers' fault-fixing efforts, using the three measures defined in the previous section. The first research question is a preliminary question about the relation between design smells and faults. It allows us to collect required data for the subsequent RQs and is a replication of a previous work [Khomh *et al.*, 2011a]. Thus, it plays the role of a sanity check before addressing the four novel RQs. The remaining four research questions pertain to fault-fixing duration, impacted elements, their entropies, and (conversely) the design smells impacted by fault fixes.

For the preliminary research question, we formulate two hypotheses, pertaining to systems as whole or to each of their versions individually (H_{01A} and H_{01V}). For each of the last subsequent questions, we formulate two hypotheses: one concerning all the design smells collectively ($H_{02A} \ldots H_{05A}$) and another one concerning the impact of each kind of design smell independently ($H_{02K} \ldots H_{05K}$).

5.2.1 Research Questions

5.2.1.1 Relation between Design Smells and Faults

First, we ask a preliminary research question about the relation between faults and design smells. This preliminary question addresses the relation between the number of occurrences of design smells in a class and the number of occurrences of faults in that class and ask: **RQ1: What is the relation between the number of design smells in a class and its fault-proneness?**

To answer this question, we analyse whether classes participating in a higher number of occurrences of design smells are more fault-prone than other classes, by testing two null hypotheses:

- H_{01A}: The number of occurrences of design smells in fault-prone classes is not significantly higher than the number of occurrences of design smells in other classes across all versions of a system when considering all the classes in all the versions of a program as one dataset

- H_{01V}: The number of occurrences of design smells in fault-prone classes is not significantly higher than the number of occurrences of design smells in other classes when considering each version of a program independently.

We test these two hypotheses to observe both a "global" (across all versions) and a "local" effect (version by version) of design smells on fault proneness. We must observe these two effects to ensure that the impact is not circumstantial (in few given versions but not in all versions) but is also not due to some other, unknown factors (in all versions but not in a particular version).

5.2.1.2 Relation between Design Smells and the Duration of Fixing Periods

Second, we study the relation between faults involving classes with design smells and the duration of fixing periods and ask: **RQ2: What is the relation between the duration of fixing periods and the participation in design smells of classes involved in the corresponding faults?**

We are interested to evaluate whether developers take more time to fix faults involving classes with design smells than faults involving classes without design smells, by testing the null hypothesis:

- H_{02A}: There is no statistically-significant difference between the duration of fixing periods for faults involving classes with design smells and that for faults involving classes without design smells.

We are also interested to evaluate whether faults involving classes with particular kinds of design smells take more time to be fixed than other faults, by testing the null hypothesis:

- H_{02K}: There is no statistically-significant difference between the duration of fixing periods for faults involving classes with a particular kind of design smells and that for other faults.

5.2.1.3 Relation between Design Smells and the Number of Impacted Elements

Third, we address the relation between faults involving classes with design smells and the number of elements (fields and methods) of the classes impacted by the fault-fixing and ask: **RQ3: What is the relation between the number of elements (fields and methods) of the classes impacted by fault-fixes and the participation in design smells of classes involved in the corresponding faults?**

We analyse whether the fixes of faults involving classes with design smells impact more elements than that of faults involving classes without design smells, by testing the null hypothesis:

- H_{03A}: There is no statistically-significant difference between the number of elements impacted by the fixes of faults involving classes with design smells and that of faults involving classes without design smells.

We also analyse whether the fixes of faults involving classes with particular kinds of design smells impact more elements than fault-fixes of other faults, by testing the null hypothesis:

- H_{03K}: There is no statistically-significant difference between the number of elements impacted by the fixing of faults involving classes with a particular kind of design smells and that of other faults.

5.2.1.4 Relation between Design Smells and Fault-fixing Entropy

Fourth, we address the relation between faults involving classes participating in design smells and the entropy of fault-fixes and ask: **RQ4: What is the relation between the entropy of fault-fixes and the participation in design smells of classes involved in the corresponding faults?**

We analyse whether the fixes of faults involving classes with design smells have higher entropy than that of faults involving classes without design smells, by testing the null hypotheses:

- H_{04A}: There is no statistically-significant difference between the entropy of the fixes of faults involving classes with design smells and that of faults involving classes without design smells.

We also analyse whether the fault-fixes of faults involving classes with particular kinds of design smells have higher entropy than fault-fixing of other faults, by testing the null hypothesis:

- H_{04B}: There is no statistically-significant difference between the entropy of the fixes of faults involving classes with a particular kind of design smells and that of other faults.

5.2.1.5 Relation between Fault-fixing and the Number of Occurrences of Design Smells

Fifth, we address the impact of fault-fixing on the number of occurrences of design smells and ask: **RQ5: What is the relation between fault-fixes and the number of occurrences of design smells in classes involved in faults before and after fixing the faults?**

We analyse whether fault-fixes impact the number of occurrences of design smells in classes involved in faults, by testing the null hypothesis:

- H_{05A}: There is no statistically-significant difference between the number of occurrences of design smells in classes involved in faults before fixing these faults and that in classes involved in faults after fixing the faults.

We also analyse whether fault-fixes impact classes with particular kinds of design smells differently that other classes, by testing the null hypothesis:

- H_{05K}: There is no statistically-significant difference between the number of occurrences of a particular kind of design smells in classes involved in faults before fixing these faults and that after fixing the faults.

5.2.2 Independent Variables

In our study, we associate to each class a set of variables $AP_{i,j,r}$, which indicate the number of occurrences of a design smell j in which a class i participates in a release r of a system under study. We name R_s the set of releases for a system under study and D the set of the 12 studied design smells. S_t is the set of classes involved in the fix of a fault t.

We use these variables in the formulation of the independent variables as follows:

- Relation between Design Smells and Faults

  In H_{01A}, we consider all the classes in all the versions of a system as one dataset. The independent variable, $HasF_i$, is a Boolean variable that indicates whether a class i underwent or not at least one fault-fix between the dates of the first and the last studied releases of the system.

  In H_{01V}, we analyse the relation between the number of occurrences of design smells in which a class participates and its fault-proneness, version per version independently. The independent variable, $HasF_{i,j}$, is a Boolean variable that indicates whether a class i underwent at least one fault-fix between the studied version j of a system and its version $j+1$.

- Relation between Design Smells and the Duration of Fixing Periods, Relation between Design Smells and the Number of Impacted Elements, and Relation between Design Smells and Fault-fixing Entropy

  In H_{02A}, H_{03A}, and H_{04A}, the independent variable, $HasS_t$, is a Boolean variable that indicates whether a fault t involves at least one class participating to at least one occurrence of some design smells. We formulate that:

$$HasS_t = \begin{cases} 0, & \sum_{i \in S_t, j \in D} AP_{i,j,r} = 0; \\ 1, & \sum_{i \in S_t, j \in D} AP_{i,j,r} > 0. \end{cases}$$

In H_{02K}, H_{03K}, and H_{04K}, the independent variable, $AP_{j,r}$, indicates the number of occurrences of a particular design smell j in the classes of S_t involved in the fix of a fault t reported against the version r of a system. We write that:

$$AP_{j,r} = \sum_{i \in S_t, j \in D} AP_{i,j,r}$$

- Relation between Fault-fixing and the Number of Occurrences of Design Smells

  In H_{05A}, the independent variable, $AP_{t,c}$, indicates the number of occurrences of the design smells in the set S_t of classes involved in a fault t in the version r in which the fault t has been reported (*i.e.*, before the fault-fix). We formulate:

$$AP_{t,r} = \sum_{i \in S_t, j \in D} AP_{i,j,r}$$

  In H_{05K}, for each kind of design smell k, the independent variable, $AP_{t,c,k}$, indicates the number of occurrences of design smell of type k in the set S_t of classes involved in a fault t in the release r in which the fault t has been reported (*i.e.*, before the fault-fix).

  We formulate:

$$AP_{t,r,k} = \sum_{i \in S_t} AP_{i,k,r}$$

5.2.3 Dependent Variables

Following the notations in the previous subsection, we formulate the dependent variables for the different hypotheses. Some dependent variables for some RQs are independent variables of other RQs.

- Relation between Design Smells and Faults

  In H_{01A}, we consider all the classes in all the versions of a system as one dataset. The dependent variable, AP_i, indicates the total number of occurrences of design smells in which a class i participates. We formulate that:

$$AP_i = \sum_{j \in D, k \in R_s} AP_{i,j,k}$$

In H_{01V}, we analyse each version independently. The dependent variable, $AP_{i,r}$, indicates, for a studied version r, the total number of occurrences of design smells in which a class i participates. We formulate that:

$$AP_{i,r} = \sum_{j \in D} AP_{i,j,r}$$

- Relation between Design Smells and the Duration of Fixing Periods

  For H_{02A} and H_{02K}, the dependent variable D_t, duration of the fixing period, indicates the number of days between the date of the report of the fault t ($D_{r,t}$) and the day of the fix or correction ($D_{c,t}$) of t. We formulate that:

$$D_t = D_{c,t} - D_{r,t}$$

- Relation between Design Smells and the Number of Impacted Elements

  For H_{03A} and H_{03K}, the dependent variable is the number of elements Nbr_{Elts} impacted by a fault-fix. We define Nbr_{Elts} as the total number of methods and fields that the developers modified to fix a fault. $N_{m,t}$ and $N_{f,t}$ are, respectively, the number of methods and fields added/removed to fix the fault t. We compute $N_{f,t}$ and $N_{m,t}$ by parsing the patch diff files to count the number of added/removed fields and the number of added/removed methods.

  We formulate:

$$Nbr_{Elts} = N_{m,t} + N_{f,t}$$

- Relation between Design Smells and Fault-fixing Entropy

  For H_{04A} and H_{04K}, the dependent variable, E_t, is the entropy of the fix of a fault t. The entropy of a fault-fix represents the complexity of that fault-fix and reflects the disorder spread by that fault-fix in the source code. We recall that:

$$E_t = -\sum_{i=1}^{n} \left(\frac{l_i}{l} \times log_n \frac{l_i}{l}\right),$$

- Relation between Fault-fixing and the Number of Occurrences of Design Smells

We study the relation between the number of occurrences of design smells in the classes involved in a fault t before the fix of t and that after the fix of t. In H_{05A}, the dependent variable, $AP_{t,c}$, indicates the number of occurrences of design smells in the set S_t of classes involved in a fault t after the correction of the fault (*i.e.*, after the fault-fix).

We formulate:

$$AP_{t,c} = \sum_{i \in S_t, j \in D} AP_{i,j,c}$$

In H_{05K}, for each kind of design smell k, the dependent variable, $AP_{t,c,k}$, indicates the number of occurrences of design smell of type k in the set S_t of classes involved in a fault t after the correction of the fault (*i.e.*, after the fault-fix). We formulate:

$$AP_{t,c,k} = \sum_{i \in S_t} AP_{i,k,r}$$

5.2.4 Analysis Method

We divide our analysis method in four specific analyses for:

- H_{01A} and H_{01V};
- H_{02A}, H_{03A}, and H_{04A};
- H_{02K}, H_{03K}, and H_{04K};
- H_{05A} and H_{05K}.

5.2.4.1 H_{01A} and H_{01V}

We compute the (non-parametric) Mann-Whitney test to compare the two groups of classes, those with faults and those without faults in H_{01A} and H_{01V}, with respect to their number of occurrences of design smells and analyse whether the difference in their average numbers of occurrences of design smells is statistically significant. We use the Mann-Whitney test because, as a non-parametric test, it does not make any assumption on the underlying distribution, see Section 2.5.2.

In H_{01A}, we analyse the results of the test to see if the difference is statistically significant. Results are intended as statistically significant at $\alpha = 0.05$. Therefore, if p-value < 0.05, we reject the null hypothesis H_{01A} and conclude that the average number of occurrences of design smells in fault-prone classes is higher than that in non fault-prone classes.

We will reject the hypothesis H_{01V} if the p-values are $< \alpha$ for each version for 75% of the analysed versions, as in previous work [Khomh *et al.*, 2011a]. We will then conclude that, per version, the average number of occurrences of design smells in higher in fault-prone classes.

Other than testing the hypotheses, we also estimate the magnitude of the difference in means between the two groups. We use the non-parametric effect size measure Cliff's d, which indicates the magnitude of the effect size of the treatment on the dependent variable, see Section 2.5.4.

5.2.4.2 H_{02A}, H_{03A}, and H_{04A}

We compute the (non-parametric) Mann-Whitney test to compare the group of faults involving classes with design smells and the group of faults involving classes without design smells. We analyse the results to see whether their difference is statistically significant, again with $\alpha = 0.05$.

Therefore, for H_{02A}, if p-value $< \alpha$, we reject the null hypothesis and conclude that the average duration of fixing periods of faults involving classes with design smells is higher than that of faults involving classes without design smells.

For H_{03A}, if p-value $< \alpha$, we reject the null hypothesis and conclude that the average number of elements impacted by the fixes of faults involving classes with design smells is higher than that of faults involving classes without design smells.

For H_{04A}, if p-value $< \alpha$, we reject the null hypothesis and conclude that the average fault-fixing entropy of faults involving classes with design smells is higher than that of faults involving classes without design smells.

5.2.4.3 H_{02K}, H_{03K}, and H_{04K}

For the hypotheses H_{02K}, H_{03K}, and H_{04K}, we analyse the relation between particular kinds of design smells and:

- the durations of the fixing periods (H_{02K});
- the number of elements (fields, methods) impacted by fault-fixes (H_{03K});
- the entropy of the fault-fixes (H_{04K}).

To determine the weight of each kind of design smells in numbers, we use multivariate regression analyses because one of the uses of regression analyses is to understand which, among some independent variables, *i.e.*, the different kinds of design smells, are related to the dependent

variables, *i.e.*, the durations of fixing periods, the number of elements impacted by fault-fixes, and the entropy of fault-fixes.

The general linear regression model with p independent variables is based on the formula:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i$$

where:

- x_{ij} are characteristics describing the modelled phenomenon, in our case, the number of each kind j of design smells involved in the fixing of fault i;
- β_j are the model coefficients.

Based on the p-values of each variable x_{i1}, x_{i2}, ..., x_{ip}, we reject the null hypotheses H_{02K}, H_{03K}, and H_{04K}, if the p-value of the variable x_{ip} that describes a particular kind of design smells is less than α. If p-value $< \alpha$ then we assume that the particular design smell impacts more the developers' fault-fixing effort than others.

5.2.4.4 H_{05A} and H_{05K}

For H_{05A} and H_{05K}, the two groups of classes, before and after the fault-fixes, are not independent. Thus, we use the paired Mann-Whitney test for the two groups, before and after fault-fixes, to compare the number of occurrences of design smells in each group. We analyse the results to see whether the difference is statistically significant. We again choose $\alpha = 0.05$. If p-value $< \alpha$, we reject the null hypothesis H_{05A} and conclude that the average number of occurrences of design smells increases or decreases with the fault-fixes.

For H_{05K}, we do the same computation as in H_{05A} for each particular kind of design smell and, in the same condition, we conclude on the effect of fault-fixes on the number of occurrences of each particular kind of design smell.

5.3 Study Results

We now report the results of the analyses performed to answer the research questions.

Table 5.1 – Mann-Whitney tests and Cliff's d results for the number of occurrences of design smells in classes with faults compared to classes without faults across all versions.

Systems	M-W p	Cliff's d
ArgoUML	$<$ **0.01**	0.55
Eclipse	$<$ **0.01**	0.28
Mylyn	$<$ **0.01**	0.13
Rhino	$<$ **0.01**	0.45

5.3.1 Relation between Design Smells and Faults

Table 5.1 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the number of occurrences of design smells in classes with faults and that in classes without faults, across all versions.

For the four studied systems, the results show that there is statistically significant difference between the number of occurrences of design smells in classes with faults and that without faults with small effect size for Mylyn, medium effect size for Eclipse, and large effect size for ArgoUML and Rhino.

We thus can reject the null hypothesis H_{01A} and conclude that the number of occurrences of design smells in classes with faults is significantly higher than that in other classes across all versions with large effect size for two studied systems and medium and small effect size for the other two studied system.

Table 5.2 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the number of occurrences of design smells in classes with faults and that in classes without faults, per version. For some releases, the tests could not be performed (N/A) because of the small numbers of faults and–or design smells.

For the analysed versions of the four studied systems, the p-values in 36 out of 39 versions are statistically significant for H_{01V} with a small effect size for 9 versions (25%), medium effect size for 8 versions (22%), and large effect size for 13 versions (36%). These results indicate that, for most of the analysed versions, fault-prone classes are those with higher numbers of occurrences of design smells.

We thus can reject the null hypothesis H_{01V} and conclude that the number of occurrences of design smells in classes with faults is significantly higher than that in other classes per version with large or medium effect size for most of the studied versions of the systems (21).

Table 5.2 – Mann-Whitney tests and Cliff's d results for the number of occurrences of design smells in classes with faults compared to classes without faults per version.

ArgoUML			Eclipse			Mylyn			Rhino		
Release	M-W p	Cliff's d	Release	M-W p	Cliff's d	Release	M-W p	Cliff's d	Release	M-W p	Cliff's d
0.10.1	< **0.01**	0.373	1.0	< **0.01**	0.146	1.0.1	< **0.01**	0.386	1.4R3	0.34	N/A
0.12	0.82	N/A	2.0	< **0.01**	0.085	2.0.0	< **0.01**	0.905	1.5R1	< **0.01**	0.795
0.14	< **0.01**	0.331	2.1.1	< **0.01**	0.093	2.0M1	< **0.01**	0.567	1.5R2	< **0.01**	0.864
0.16	< **0.01**	0.199	2.1.2	< **0.01**	0.158	2.0M2	< **0.01**	0.591	1.5R3	< **0.01**	0.514
0.18.1	< **0.01**	0.369	2.1.3	< **0.01**	0.208	2.0M3	0.98	N/A	1.5R4	N/A	
0.20	< **0.01**	0.446	3.0	< **0.01**	0.223	2.1	N/A		1.5R41	< **0.01**	0.778
0.22	< **0.01**	0.592	3.0.1	< **0.01**	0.179	2.2.0	N/A		1.5R5	< **0.01**	0.561
0.24	< **0.01**	0.134	3.0.2	< **0.01**	0.065	2.3.0	N/A		1.6R1	< **0.01**	0.741
0.26	< **0.01**	0.353	3.2	< **0.01**	0.173	2.3.1	N/A		1.6R2	N/A	
0.26.2	< **0.01**	0.556	3.2.1	< **0.01**	0.163	2.3.2	< **0.01**	0.442	1.6R3	N/A	
			3.2.3	< **0.01**	0.147	3.0.0	N/A		1.6R4	< **0.01**	0.529
			3.3	< **0.01**	0.234	3.0.1	N/A		1.6R5	< **0.01**	0.595
			3.3.1	< **0.01**	0.074	3.0.2	N/A		1.6R6	N/A	
						3.0.3	N/A				
						3.0.4	N/A				
						3.0.5	N/A				
						3.1.0	N/A				
						3.1.1	< **0.01**	0.462			

Table 5.3 – Mann-Whitney tests and Cliff's d results for the duration of the fixing periods.

System	M-W p	Cliff's d
ArgoUML	0.60	–
Eclipse	$<$ **0.05**	0.10
Mylyn	$<$ **0.05**	0.15
Rhino	0.87	–

5.3.2 Relation between Design Smells and the Duration of Fixing Periods

Table 5.3 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the the Mann-Whitney tests and Cliff's d effect sizes to compare the duration of the fixing periods for faults involving classes with design smells and that for faults involving classes without design smells.

We put $-$ on the Cliff's d effect sizes column to show that we do not compute the Cliff's d effect sizes when the difference is not statistically significant.

The results show that for Eclipse and Mylyn, the difference between the duration of the fixing periods for faults involving classes with design smells and that for faults involving classes without design smells is statistically significant.

For ArgoUML and Rhino, the difference between the duration of the fixing periods for faults involving classes with design smells and that for faults involving classes without design smells is not statistically significant.

> We can reject the hypothesis H_{02A} for Eclipse and Mylyn. We cannot reject H_{02A} for ArgoUML and Rino. We thus conclude that the duration of the fixing periods for faults involving classes with design smells or not are system dependent

Table 5.4 summarises the results of the multiple regression describing the relations between the duration of the fixing periods and the different kinds of design smells. We observe that the durations are not related to any particular kind of design smells because none of the particular kind of design smells impact the durations of the fixing periods significantly across all systems, *i.e.*, there is no full line of check marks.

> We cannot reject H_{02K} for any design smell in any of the studied systems. We thus conclude that the different kind of design smells can collectively impact the durations of the fixing periods for faults not individually

Table 5.4 – Kinds of design smells significantly correlated with the duration of the fixing periods.

	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton				✓
Blob				
ClassDataShouldBePrivate				
ComplexClass				
LargeClass				
LazyClass		✓		
LongMethod		✓		
LongParameterList		✓		
MessageChains		✓		
RefusedParentRequest				
SpeculativeGenerality				
SwissArmyKnife		✓		

Table 5.5 – Mann-Whitney tests and Cliff's d results for the number of elements impacted by fault-fixes.

System	M-W p	Cliff's d
ArgoUML	0.09	–
Eclipse	$<$ **0.05**	0.57
Mylyn	$<$ **0.05**	0.40
Rhino	$<$ **0.05**	0.40

5.3.3 Relation between Design Smells and the Number of Impacted Elements

Table 5.5 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the number of elements impacted by fault-fixes of faults involving classes with design smells and that of faults involving classes without design smells. We put . on the Cliff's d effect sizes column to show that we do not compute the Cliff's d effect sizes when the difference is not statistically significant.

The results show that, for Eclipse, Mylyn, and Rhino (75% of the studied systems), the difference between the number of elements impacted by the fixes of faults involving classes with design smells and that of faults involving classes without design smells is statistically significant with medium effect size for Mylyn and Rhino and with large effect size for Eclipse.

Table 5.6 – Kinds of design smells significantly correlated with the number of elements impacted by fault-fixes.

	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton		✓		✓
Blob	✓	✓		
ClassDataShouldBePrivate		✓	✓	✓
ComplexClass		✓	✓	✓
LargeClass	✓			✓
LazyClass		✓		
LongMethod	✓			✓
LongParameterList		✓	✓	✓
MessageChains		✓	✓	✓
RefusedParentRequest	✓	✓	✓	
SpeculativeGenerality		✓		
SwissArmyKnife		✓		

Therefore, we can reject the null hypothesis H_{03A} and conclude that the difference between the number of elements impacted by the fixes of faults involving classes with design smells and that of faults involving classes without design smells are statistically significant with medium and large effect size

Table 5.6 summarises the results of the multiple regression describing the relations between the number of elements impacted by fault-fixes and the different kinds of design smells.

The different kinds of design smells related to the number of elements impacted by the fault-fixes in 75% of studied systems are: ClassDataShouldBePrivate, ComplexClass, LongParameterList, MessageChains and RefusedParentRequest.

We can reject the null hypothesis H_{03B} and conclude that some particular kinds of design smells (ClassDataShouldBePrivate, ComplexClass, LongParameterList, MessageChains and RefusedParentRequest) make fault-fixes impact more fields and method than others.

5.3.4 Relation between Design Smells and Fault-fixing Entropy

Table 5.7 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the entropy of fixes of faults involving classes with design smells and that of faults involving classes without design smells.

Table 5.7 – Mann-Whitney tests and Cliff's d results for fault-fixing entropy.

System	M-W p	Cliff's d
ArgoUML	$< \mathbf{0.05}$	0.79
Eclipse	$< \mathbf{0.05}$	0.34
Mylyn	$< \mathbf{0.05}$	0.36
Rhino	$< \mathbf{0.05}$	0.25

Table 5.8 – Kinds of design smells significantly correlated with the fault-fixing entropy.

	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton				
Blob	✓	✓		
ClassDataShouldBePrivate		✓		✓
ComplexClass		✓		
LargeClass				
LazyClass		✓	✓	
LongMethod		✓	✓	
LongParameterList	✓	✓		
MessageChains				
RefusedParentRequest				
SpeculativeGenerality		✓		✓
SwissArmyKnife				

The results show that, for all the analysed systems, the difference between the entropy of fixes of faults involving classes with design smells and that of faults involving classes without design smells is statistically significant with medium and large effect sizes.

Therefore, we can reject the null hypothesis H_{04A} and conclude that the fixes of faults involving classes with design smells have higher entropies than that of faults involving classes without design smells with small effect size for Rhino, medium effect size for Eclipse and Mylyn, and large effect size for ArgoUML.

Table 5.8 summarises the results of the multiple regression describing the relations between the fault-fixing entropy and the different kinds of design smells. We observe that the fault-fixing entropy is not related to any particular kind of design smells because none of the particular kind of design smells impact the fault-fixing entropy significantly across all systems, *i.e.*, there is no full line of check marks.

Table 5.9 – Mann-Whitney results for the number of occurrences of design smells in classes before and after fault-fixes.

System	M-W p	Cliff's d
ArgoUML	$<$ **0.01**	0.52
Eclipse	$<$ **0.01**	0.22
Mylyn	$<$ **0.01**	0.15
Rhino	$<$ **0.01**	0.25

We cannot reject H_{04K} for any particular kind of design smell for all the studied systems. We thus conclude that the different kind of design smells can collectively impact the fault-fixing entropy not individually. The negiative impact of some particular kind of design smells (Blob, ClassDataShouldBePrivate, LazyClass, LongMethod, LongParameterList, SpeculativeGenerality) on fault-fixing entropy is system dependent

5.3.5 Relation between Fault-fixing and the Number of Occurrences of Design Smells

Table 5.9 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and Cliff's d effect sizes to compare the number of occurrences of design smells in classes involved in faults before and after fixing the faults.

The results show that, for all the analysed systems, the difference between the number of occurrences of design smells in classes involved in faults before and after fixing the faults is statistically significant with small effect sizes for Eclipse, Mylyn and Rhino and with large effect size for ArgoUML.

Therefore, we can reject the null hypothesis H_{05A} and conclude that, after fault-fixing, the number of occurrences of design smells decrease significantly with small and large effect sizes.

Table 5.10 reports for ArgoUML, Eclipse, Mylyn, and Rhino the results of the Mann-Whitney tests and the direction of evolution when comparing the number of occurrences of particular kinds of design smells in classes before and after fixing faults.

The results show that, for some of the different kinds of design smells in ArgoUML (7 out of 9 kinds of design smells), Eclipse (7 out of 9 kinds of design smells), Mylyn (1 out of 8 kinds of design smells), and Rhino (5 out of 7 kinds of design smells), fault-fixes impact significantly the number of occurrences of these different kinds of design smells.

Therefore, we can reject the null hypothesis H_{05K} and conclude that, after fault-fixing, the number of occurrences of some kinds of design smells decrease/increase significantly

Table 5.10 – Mann-Whitney results for the number of occurrences of design smells in classes before and after fault-fixes.

Design smells	ArgoUML		Eclipse		Mylyn		Rhino	
	p-value	Impact	*p*-value	Impact	*p*-value	Impact	*p*-value	Impact
AntiSingleton	$<$ **0.01**	–	$<$ **0.01**	–	N/A	N/A	$<$ **0.01**	–
Blob	$<$ **0.01**	+	0.64	–	0.06	+	N/A	N/A
ClassDataShouldBePrivate	1.00	0	$<$ **0.05**	+	0.98	–	0.38	–
ComplexClass	$<$ **0.01**	+	$<$ **0.01**	–	0.086	+	$<$ **0.02**	+
LargeClass	$<$ **0.01**	+	N/A	N/A	0.1	+	$<$ **0.02**	+
LazyClass	N/A	N/A	$<$ **0.01**	0	N/A	N/A	N/A	N/A
LongMethod	$<$ **0.01**	+	$<$ **0.01**	+	$<$ **0.02**	+	0.21	–
LongParameterList	$<$ **0.02**	+	0.41	–	0.86	–	$<$ **0.01**	–
MessageChains	$<$ **0.02**	+	$<$ **0.01**	–	0.12	+	$<$ **0.03**	–
RefusedParentRequest	0.09	+	$<$ **0.01**	+	0.14	+	N/A	N/A

5.4 Discussion

This section discusses the results reported in Section 5.3, summarised in Table 5.11. We structure the discussion as follows: first, we discuss statistically significant results, second, we discuss non-statistically significant results and, finally, we discuss general concerns about our experiments and results.

5.4.1 Statistically-significant Results

5.4.1.1 Relation between Design Smells and Faults

Table 5.2 shows that there are significant differences between the number of occurrences of design smells in the classes with faults and the number of occurrences of design smells in the classes without faults.

These results confirm those obtained in previous work and we can explain the relation in two ways. First, some design smells impact directly the implementation of a method/class, such as the Spaghetti Code, which is a symptom of complex logics with interlocked control flows. Thus, the occurrences of such design smells could directly lead to more complexity and, thus, more propension to faults. Second, some design smells impact indirectly the implementation of a method/class, such ClassDataShouldBePrivate: per se, occurrences of this design smell may not yield to faults but they could lead to developers mistakenly writing code that indirectly lead classes in inconsistent states.

5.4.1.2 Relation between Design Smells and the Duration of Fixing Periods

Results of RQ2 show that the duration of fixing periods increases for Eclipse and Mylyn in the cases of classes with occurrences of design smells. However, no design smell in particular satisfied our threshold of 75% of the systems. We explain these results using Table 2.1. Table 2.1 reports the number of occurrences of design smells in Eclipse and Mylyn when compared to these in ArgoUML and Rhino and shows that there are more occurrences in Eclipse and Mylyn than in the other two systems. Also, Table 2.1 reports that these numbers of design smells (in percentages of the total number of classes) seem to increase in Eclipse and Mylyn while they decrease in the other two systems. Thus, the sheer numbers of occurrences of design smells and their trends could explain the statistically-significant relations between design smells and the duration of fixing periods in Eclipse and Mylyn: more occurrences impede the developers' fault-fixing efforts.

In particular, we can explain the negative impact of the LongMethod, LongParameterList, MessageChain, and SwissArmyKnife design smells on the durations of fault-fixes as follows: these smells have in common to increase some characteristics of the source code such as the method length, the number of parameters of a method, number of conditional statements. Long methods and/or huge number of parameters could lead to difficulty to understand the code and thus, could make developers spending more effort to understand them when they need to fix them.

The relation between the AntiSingleton and LazyClass design smells and the duration of fixes could be due to the time spent by developers over classes whose design and implementation do little (LazyClass) or have unnecessary public methods (AntiSingleton). Thus, occurrences of these two design smells "sand in the way" of the developers' fixing efforts.

5.4.1.3 Relation between Design Smells and the Number of Impacted Elements and Relation between Design Smells and Fault-fixing Entropy

Table 5.5 shows a correlation between the presence of design smells in a faulty class and the number of fields and methods impacted by its fix. We observe that the fixing of faults involving classes with design smells impacts more fields and methods than others. We explain this correlation with the relation between design smells and the design and implementation of systems: design smells, even if located in one class, relate actually to several classes. For example, the ClassDataShouldBePrivate design smell implies that other classes are using the fields declared in a class directly, while these fields should be private. Thus, when developers must fix a bug in such a class, they must check all the "surrounding" classes to make sure that they do not inadvertently introduce faults and, when changing this class, must adapt all these "surrounding" classes to the changes.

In particular, fixes related to some design smells impact more fields and methods than other fixes: ClassDataShouldBePrivate, ComplexClass, LongParameterList, MessageChains and RefusedParentRequest. These design smells either impact the interactions among collaborating classes (ClassDataShouldBePrivate, LongParameterList, MessageChains, and RefusedParentRequest) or per se, imply more fields and methods (ComplexClass). Therefore, it is not surprising that occurrences of these design smells require more fields and methods to be changed when fixing faults. Hence the observed high entropy values.

5.4.1.4 Relation between Fault-fixing and the Number of Occurrences of Design Smells

We rejected the null hypothesis H_{05A} and concluded that, after fault-fixing, the number of occurrences of design smells decrease significantly with small and large effect sizes. We explain the relation between fault fixes and the decreases in the number of occurrences of design smells as follows. Because design smells increase the developers' efforts both in time spent and in the number/entropy of the impacted fields and methods, developers may remove, intentionally or not, design smells as they fix faults as a means to ease fixing, testing, and future changes. When they fix a fault, they can observe that some structures make the work difficult to achieve and they can then decide to refactor the code while fixing the fault.

More studies are required to confirm developers' intentions for removing such smells.

We also reject the null hypothesis H_{05K} and conclude that, after fault-fixing, the number of occurrences of different kinds of design smells decreases/increase significantly. For exemple fault-fixing decreases the number of :

- AntiSingleton (in ArgoUML, Eclipse, and Rhino),
- CompleClass (in Eclipse),
- LongParameterList (in Rhino),
- MessageChains (in Eclipse, and Rhino).

The fault-fixing, also increases the number of :

- Blob (in ArgoUML),
- CompleClass (in ArgoUML, and Rhino),
- LongMethod (in ArgoUML, Eclipse, and Mylyn),
- LongParameterList (in ArgoUML),

- MessageChains (in ArgoUML),
- RefusedParentRequest (in Eclipse).

Future work includes explaining these differences in observation between the different kind of design smells and among the systems.

5.4.2 Non-statistically-significant Results

5.4.2.1 Relation between Design Smells and Faults

All results for RQ1 were statistically significant.

5.4.2.2 Relation between Design Smells and the Duration of Fixing Periods

We could not reject H_{02A} for ArgoUML and Rhino. We explain the lack of relation between design smells and the duration of fault fixes by the small number of occurrences of the smells in these two systems and the small number of faults in these two systems compared to Eclipse and Mylyn.

We cannot reject H_{02K} for any particular kind of design smell in more than one of the studied system. Again, we explain the lack of relation between the different kinds of design smells and the duration of fault fixes by the small number of occurrences of the different kinds of design smells in these two systems and the small number of faults.

5.4.2.3 Relation between Design Smells and the Number of Impacted Elements and Relation between Design Smells and Fault-fixing Entropy

We cannot reject H_{04K} for any particular kind of design smell in more than 50% of the studied systems. We explain this lack of clear correlation by the proportion of the different kinds of design smells in the various systems. Future work includes performing more experiments on other systems to confirm/infirm the observations, see also Section 5.5.

5.4.2.4 Relation between Fault-fixing and the Number of Occurrences of Design Smells

All results for RQ5 were statistically significant.

5.4.3 General Discussions

An important threat to the validity of our study is the difference between the versions in which a fault may be introduced/fixed and that in which it is reported as occurring and as fixed. Indeed, it is possible that a fault exists prior to the version against which it is reported, just because it has never been encountered in previous versions or because users did not bother reporting it in previous versions. Similarly, it is possible that a fault is fixed in a version prior to that for which it is reported as fixed due to the time to test the code thoroughly or because of an omission to do so.

We use the statistically-significant results of RQ1 and RQ5 to warn developers about the negative impact on faults of design smells (RQ1). Developers should thus be wary of and track design smells. Because the number of occurrences of design smells is related to the presence of faults in a class, correcting these design smells may help avoiding or reducing the risk of faults in classes while removing fault reduces the number of smells (RQ5).

5.5 Threats to validity

We now discuss the threats to the validity of our study following the guidelines for case study research [Yin, 2008].

5.5.1 Construct validity

Construct validity threats concern the relation between theory and observation. In our study, they are mainly due to measurement errors. The identification of classes impacted by faults, fault-fixes requires analysing issue-tracking systems and version-control systems. For Rhino, we use an existing classification [Eaddy *et al.*, 2008]. For the other three systems, we made sure that our algorithms provided correct results by veryfying manually random samples.

For design smells detection, we use DECOR, which includes its authors' subjective understanding of the design smells. The accuracy of its detection algorithms is not perfect [Moha *et al.*, 2010a]. DECOR accuracy impacts our results because we may have classified a class not participating in a design smell as participating in it and vice-versa. Other techniques and tools should be used to confirm our results.

5.5.2 Internal Validity

Threats to internal validity do not affect our study, being an exploratory study [Yin, 2008]. We do not claim causation, but relate the presence of design smells with fault-fixing durations, the number of elements impacted by fault-fixes, and fault-fixing entropy.

5.5.3 Conclusion Validity

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. We used only non-parametric tests that do not make any assumption about the data set distribution (*i.e.*, Mann-Whitney tests and Cliff's *d* effect size).

5.5.4 Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. Moreover, the source code repositories and issue-tracking systems of the studied systems are available to obtain the same data. Moreover, the results of the validation and the datasets are available on-line [1].

5.5.5 External Validity

Threats to external validity concern the possibility to generalise our results. We studied four software systems having different sizes and belonging to different domains. We used also a representative subset of design smells. Future work includes reproducing this study with other software systems and other design smells.

5.6 Summary

We posed the following premise:

There is a relation between design smells, faults, and the developers' effort to fix faults.

We performed an empirical study and analysed: (1) the relation between design smells and faults; (2) the impact of the presence of design smells in faults on fault-fixing duration; (3) the impact of the presence of design smells in faults on the number of fields and methods impacted by

[1] http://www.ptidej.net/download/experiments/emse12/

fault-fixing; (4) the impact of the presence of design smells in faults on fault-fixing entropy; and, (5) the impact of fault-fixing on the number of occurrences of design smells.

We performed our study on 54 versions of ArgoUML, Eclipse, Mylyn, and Rhino and showed the negative impact of design smells on fault-fixing effort. Indeed, we provided quantitative evidences to the conjecture that design smells may make program maintenance harder by showing that faults involving classes participating in design smells: (1) take more time to fix, (2) impact more fields and methods, and (3) have higher entropy than others.

For researchers, our study brought evidence that design smells impact their fault-fixing efforts. Our results also support a posteriori the removal of design smells as early as possible from systems and, therefore, the importance and usefulness of design smells detection techniques and the use of detection tools.

For practitioners, we provided concrete evidence that developers should pay attention to the presence of design smells in systems and try to avoid them or, at least, correct them before performing maintenance tasks. Design smells increase developers' fault-fixing efforts and, consequently, may increase software maintenance costs. Also, a tester could decide to focus on classes participating in design smells, because she knows that such classes are likely to contain faults [Khomh *et al.*, 2011a]. Finally, a manager could use design-smell detection techniques to assess the volume of classes participating in design smells in a to-be-acquired system and, thus, adjust her offer and forecast the system cost-of-ownership and–or plan for refactoring because the design smells will impact fault-fixes effort.

Table 5.11 – Summary of the statistically-significant results of our study.

			ArgoUML	Eclipse	Mylyn	Rhino
RQ1	H_{01A}		✓	✓	✓	✓
	H_{01V}		✓	✓	✓	✓
RQ2	H_{02A}			✓	✓	
	H_{02K}	AntiSingleton				✓
		LazyClass		✓		
		LongMethod		✓		
		LongParameterList		✓		
		MessageChains		✓		
		SwissArmyKnife		✓		
RQ3	H_{03A}			✓	✓	✓
	H_{03K}	AntiSingleton		✓		✓
		Blob	✓	✓		
		ClassDataShouldBePrivate		✓	✓	✓
		ComplexClass		✓	✓	✓
		LargeClass	✓			✓
		LazyClass		✓		
		LongMethod	✓			✓
		LongParameterList		✓	✓	✓
		MessageChains		✓	✓	✓
		RefusedParentRequest	✓	✓	✓	
		SpeculativeGenerality		✓		
		SwissArmyKnife		✓		
RQ4	H_{04A}		✓	✓	✓	✓
	H_{04K}	Blob	✓	✓		
		ClassDataShouldBePrivate		✓		✓
		ComplexClass		✓		
		LazyClass		✓	✓	
		LongMethod		✓	✓	
		LongParameterList	✓	✓		
		SpeculativeGenerality		✓		✓
RQ5	H_{05A}		✓	✓	✓	✓
	H_{05K}		✓	✓	✓	✓

Chapter 6

SMURF: A SVM-based, Incremental Design Smells Detection Approach

In the following, we provide details on an iterative and incremental design smell detection approach, SMURF. We explain the details of SMURF and present the results of an empirical study.

Contents

6.1 Context and Problem **74**
6.2 SMURF Process **76**
6.3 Empirical Study **78**
6.3.1 Research Questions 78
6.3.2 Objects 80
6.3.3 Subjects 80
6.3.4 Data Collection and Oracles 81
6.3.5 Analysis Methods 82
6.4 Study Results **83**
6.4.1 Accuracy of SMURF Compared to That of DETEX 83
6.4.2 Accuracy of SMURF Compared to That of BDTEX 86
6.4.3 Accuracy of SMURF in Inter-systems Configurations 87
6.4.4 Accuracy of SMURF using Users' Feedback 88
6.5 Discussion **88**
6.5.1 SMURF vs. DETEX 88
6.5.2 SMURF vs. BDTEX 90
6.5.3 Other Design smells/Systems 90

6.5.4 Practitioners' Feedback 91
6.5.5 Software Evolution 91
6.5.6 SMURF and Practitioners 92
6.6 Threats to Validity **92**
6.7 Summary **93**

6.1 Context and Problem

Developers continuously evolve systems to implement and adapt to new customers' needs, as well as to fix bugs. Due to the time-to-market, lack of understanding, and the developers' experience, developers cannot always follow standard designing and coding techniques, *i.e.*, design patterns [Fowler, 1999a]. Consequently, design smells creep up in software systems.

Motivation. Researchers have performed empirical studies to show that design smells like Spaghetti Code and Blob create hurdles during program comprehension [Abbes *et al.*, 2011], software evolution and maintenance activities [Khomh *et al.*, 2011a]. The Spaghetti code design smell is characteristic of procedural thinking in object-oriented programming [Brown *et al.*, 1998]. Spaghetti code is related to classes without object-oriented structure. In general, classes are spaghetti code when they contain methods that are very long without having parameters, but with lots of branching statements. Therefore, such classes do not exploit object-oriented mechanisms, such as polymorphism and inheritance, and also prevent them from being used by developers [Moha *et al.*, 2009]. Another example of design smell is the Blob. A Blob is a large class that mostly controls the behaviour of a system or part thereof [Brown *et al.*, 1998]. The main characteristics of a Blob class are: large size, low cohesion, some method names recalling procedural programming, and association with several data classes, which only hold data with little behaviour. These two design smells and many more yield to design and implementation of systems that are, at best, cumbersome and hinder evolution, in particular by impeding program comprehension [Abbes *et al.*, 2011]. It is important to detect design smells in order to refactor/or remove them. This will improve software quality and reduce maintenance costs.

Limitations. Current design smell detection approaches as proposed by Marinescu [Marinescu, 2004b], Moha et *al.* [Moha *et al.*, 2009] and Alikacem et *al.* [Alikacem and Sahraoui, 2006], have four limitations: (1) they require extensive knowledge of design smells, (2) they have limited precision and recall, (3) they are not incremental and iterative, and (4) they cannot be applied on subsets

of systems. We claim that the first and second limitations, *i.e.*, high complexity and low accuracy, would be reduced by taking into account the practitioner's feedback. Indeed, practitioners using the textual description of design smells can easily recognise them (and then provide feedback on the detection results) but they will require more knowledge to define and set-up rules and thresholds, based on design or program metrics and properties to characterise design smells. Also, the current approaches are not flexible and cannot take advantage of practitioners feedback, so their accuracy can be improved only by changing the approach itself. Using practitioners feedback is an easy and valuable way to improve the accuracy of design smells detection.

Further, the third and fourth limitations are even more important, because they prevent a practitioner to guide the detection process and that today's software systems often weight hundreds of millions of lines of code. With such large systems, a detection approach cannot be applied frequently because of parsing and analysis times. Thus, it is important to detect design smells iteratively and incrementally to reduce the maintenance costs and encourage practitioners in the design smells detection. Indeed, if developers detect design smells independently among few classes of a software (subset of the system) as they developed, the detection will take less time. This will also permit the easy removal/or refactoring of the design smells compared to, when they perform the detection on the whole system at the end of the development. Moreover, the improvement at this stage will facilitate the next steps in the development. Also, the number of detected occurrences when dealing with the whole system can be huge and discourage the practitioners to analyse and correct them. We argue that these limitations could be taken care of by using support vector machines (SVM).

Answer. SVM have been applied in various areas, *eg.*, bioinformatics [Bedo *et al.*, 2006], information retrieval [Ye *et al.*, 2011] and object recognition [Choi *et al.*, 2012]. SVM is a recent alternative solution to the classification problem and relies on the existence of a linear classifier in an appropriate space by using a set of training data to train the parameters of the model. It is based on the use of functions called kernel, which allows an optimal separation of data in different categories. When apply to design smells detection, we believe that SVM can yield better precision and recall values when compared to that of previous approaches, and can take into account practitioners' feedback. Also, SVM is by definition incremental because we can increase its training set incrementally. Finally, it can be applied on subsets of systems because it considers system classes one at a time, not collectively as previous rule-based approaches do. To the best of our knowledge, researchers have not yet studied the potential benefits of using SVM to detect design smells.

Contribution. The contribution of this study is two-fold. First, we propose our iterative and incremental approach SMURF to detect design smells using SVM and practitioners' feedback.

We exploit the benefits of SVM to detect the occurrences of design smells while taking into account practitioners' feedback. We use the most studied design smells, *i.e.*, Blob, Spaghetti Code, Functional Decomposition, and Swiss Army Knife, and perform more than 300 experiments to compare the results of DETEX [Moha *et al.*, 2009] and BDTEX [Khomh *et al.*, 2011b], the two best state of the art approaches, respectively, in exact and probabilistic design smells detections, with the results of SMURF. We use both the measures of precision and recall to compare the approaches on a set of three programs namely ArgoUML v0.19.8, Azureus v2.3.0.6, and, Xerces v2.7.0. We showed that the accuracy of SMURF is greater than that of DETEX and BDTEX when detecting design smell occurrences on a set of classes or on the whole system. We also showed that SMURF can be applied in both intra-system and inter-system configurations. Finally, we reported that SMURF accuracy improves when using practitioners' feedback. We thus conclude that our conjecture is correct: a SVM-based approach can overcome the four limitations of previous approaches.

6.2 SMURF Process

SMURF is based on Support Vector Machines (SVM) to detect occurrences of design smells. We provide some backgrounds on SVM in Section 2.4.

Our approach to detect design smells, SMURF, is based on a SVM using a polynomial kernel, and can take into account practitioners' feedback. We use SMURF to detect the well-known design smells: Blob, Functional Decomposition, Spaghetti code, and Swiss Army Knife. For each design smell detection, the detection process is identical. Figure 6.1 shows the overview of SMURF, which we illustrate with the Blob design smell for the sake of clarity. We define:

- $TDS = \{C_i, i = 1, \ldots, p\}$, a set of classes C_i derived from an object-oriented system that constitute the training dataset;
- $\forall i, C_i$ is labelled as Blob (B) or not (N);
- DDS is the set of the classes of a system in which we want to detect Blob occurrences.

To detect the Blob classes in the set DDS, we apply SMURF through the following steps:

Step 1 (Object Oriented Metric Specification) SMURF takes as input the training dataset TDS. For each class from TDS, we calculate object-oriented metrics that will be used as the attributes x_i for each class in TDS. We use POM[1] to compute metrics for all the studied systems.

[1] `http://wiki.ptidej.net/doku.php?id=pom`

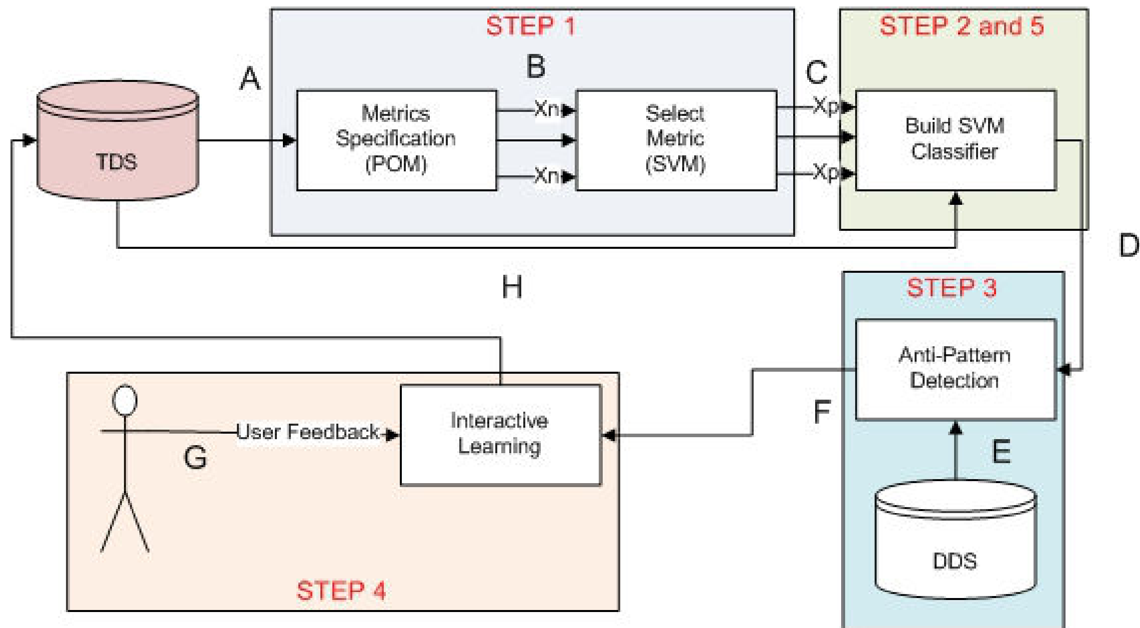


Figure 6.1 – SMURF process overview.

POM is an extensible framework, based on the PADL meta-model [Guéhéneuc and Antoniol, 2008], which provides more than 60 metrics [Guéhéneuc *et al.*, 2004], including the well-known metrics by Chidamber and Kemerer.

Step 2 (Train the SVM Classifier) We train the SVM classifier using the dataset TDS and the set of metrics computed in Step 1. We define the training dataset as: $TDS = \{(x_i, y_i) | x_i \in \mathbb{R}^p, y_i \in \{-1, 1\}, \forall i \in (1, \ldots, n)\}$ where y_i is either 1 or -1, indicating respectively if a class x_i is a Blob occurrence or not. Each x_i is a p-dimensional real vector with p the number of metrics.

The objective of the training step is to find the maximum-margin hyperplane that divides the classes into the two different groups, Blob or Not-Blob.

Step 3 (Construction of the dataset DDS and detection of the occurrences of a design smell) We build the dataset of the system on which we want to detect the occurrences of design smell as follows: for each class of the system, we compute the same set of metrics as in Step 1. We use the SVM classifier trained in Step 2 to detect the new occurrences of the design smell in the dataset *DDS*.

Step 4 (Interactive learning and practitioners' feedback) After detecting occurrences of a design smell using SMURF, the result is showed to the practitioner. Then it gives the occurrences of Blob and Non-Blob classes to the practitioner and asks for her opinion. She then can mark the occurrences as true when she agrees or false when she disagrees. We take into account the opinion of the practitioner and add her evaluated occurrences to the training dataset. This addition will permit the SVM to build a new optimal hyperplane that provides more accurate results in subsequent detections, from the practitioners' point of view. Practitioners can repeat Steps 2 to 4 as many times as desired.

6.3 Empirical Study

The *goal* of our empirical study is, by comparing our approach, SMURF with DETEX [Moha *et al.*, 2009] and BDTEX [Khomh *et al.*, 2011b], to validate that SMURF can overcome the four limitations (mentioned before) of the previous approaches. The *quality focus* of our study is the accuracy of SMURF, in terms of precision and recall. The *perspective* is that of researchers and practitioners interested in verifying if SMURF can be effective in detecting various kinds of design smells, in taking into account feedback, and in overcoming the previous limitations.

6.3.1 Research Questions

The goal of our empirical study is to evaluate the accuracy of SMURF and the impact of practitioners' feedback on the design smells detection results. We also seek to compare SMURF with DETEX [Moha *et al.*, 2009] and BDTEX [Khomh *et al.*, 2011b], the best two state of the art approaches, respectively, in exact and probabilistic design smells detection. DETEX and SMURF are both exact design smells detection approaches. Thus, we could perform a full comparison between the two approaches using the same set of programs used by Moha et *al.* [2009] in their experiment. In the case of BDTEX, which is a probabilistic design smells detection approach, we use the data provided by the authors to perform a comparison of BDTEX and SMURF.

Like other machine learning techniques, applying SMURF requires preprocessing. Indeed, we must first train SMURF on some sets of known occurrences of the design smells and non design smells, one design smell at a time, before applying it on some sets of classes. Our experiment is divided in four steps.

First, we train SMURF on a set of known occurrences of one design smell and non-design smell classes. Then, we apply SMURF on a set of classes of design smells and non-design smells occurrences. The selection of design smells and non-design smells classes is random. We also

apply DETEX on the same set of classes (we reproduce this experiment four times: as many times as the number of design smells). The first set of experiments allows us to show whether SMURF overcomes the first, second, and fourth limitations.

Second, we repeat the fist step with different sizes of subsets of the systems. Finally, we use the entire subject system. We also apply DETEX on the same subsets. These experiments allow us to compare the precision and recall of SMURF on different subset sizes with that of DETEX in a realistic settings to verify whether SMURF overcomes the first, second, and fourth limitations.

Third, we train SMURF on a set of known occurrences of one design smell and non-design smells from one system and then apply it on the classes of another system. We then add feedback to SMURF in the form of additional known occurrences of the design smell. The last set of experiments allows us to compare the accuracy of SMURF with/without feedback and the impact of practitioners' feedback on its accuracy to verify whether SMURF overcomes the first, second, and third limitations.

Finally, we train and apply SMURF on the same data that was used by BDTEX for training and testing [2] . We then compare the results of the two approaches.

Thus, we address the following research questions:

- RQ1: How does the accuracy of SMURF compare with that of DETEX, in terms of precision and recall? We decompose RQ1 as follows:
  - $RQ1_1$: How does the accuracy of SMURF compare with that of DETEX, in terms of precision and recall, when applied on a same subset of a system?
  - $RQ1_2$: How does the accuracy of SMURF compare with that of DETEX, in terms of precision and recall, when applied on a same entire system?
- RQ2: How does the accuracy of SMURF compare with that of BDTEX, in terms of precision and recall when applied on a same entire system?
- RQ3: How does the accuracy of SMURF change when trained/applied on the same system and trained/applied on different systems, in terms of precision and recall?
- RQ4: How does the accuracy of SMURF, with relevance feedback, compare with that of SMURF without feedback, in terms of precision and recall?

[2] http://www.ptidej.net/download/experiments/jss10/

Table 6.1 – Description of the objects of the study.

Names	Versions	# Lines of Code	# Classes	# Interfaces
ArgoUML	0.19.8	113,017	1,230	67
A design tool UML				
Azureus	2.3.0.6	191,963	1,449	546
A peer-to-peer client that implements the protocol BitTorrent				
Xerces	2.7.0	71,217	513	162
A syntaxic analyser				

6.3.2 Objects

The objects[3] of our study are ArgoUML v0.19.8, Azureus v2.3.0.6, and Xerces v2.7.0, three open-source Java systems. We chose these systems due to several factors. First, we selected open-source systems that are freely available so that other researchers can replicate our study. Second, we selected systems that have been used by other researchers to allow comparisons [Moha *et al.*, 2009].

Table 6.1 provides the details of the studied systems. Moha et *al.* [2009] used these three systems, thus we use these three systems for the comparison between DETEX and SMURF. Khomh et al [2011b] used only one of the three systems (Xerces v2.7.0) and thus, we compare BDTEX and SMURF using that system.

6.3.3 Subjects

The subjects of our study are the following four design smells: Blob, Functional Decomposition (FD), Spaghetti Code (SC), and Swiss Army Knife (SAK). We chose these four design smells because they are well known and commonly studied design smells and also for the comparison purpose with Moha et *al.* [2009] and Khomh et *al.* [2011b]. Indeed, Moha et *al.* [2009] used these four design smells and Khomh et *al.* [2011b] three of them.

Blob. A Blob, also called God class [Riel, 1996], is a class that is too large, with many attributes and methods, and not cohesive enough. The blob monopolises most of the processing performed by the system, takes most of the decisions, or a part thereof, directs closely the treatment of other classes, and depends on associated data classes. A data class declares only attributes and performs no processing on them.

[3] `argouml.tigris.org/`, `azureus.sourceforge.net/`, and `xerces.apache.org/xerces-c/`

Functional Decomposition. A Functional Decomposition occurs if a developer, with good experience in procedural programming and not enough knowledge of object-oriented principles, implements an object-oriented system. Brown describes this design smell as a "main routine that calls several subroutines" [Brown, 1996]. A Functional Decomposition typically yields a main class in which inheritance and polymorphism are not used.

Spaghetti Code. Spaghetti Code is an design smell that is also characteristic of procedural thinking in object-oriented programming. The Spaghetti Code is revealed by classes without structure, declaring many methods with no parameters but sharing class/instance variables. The Spaghetti Code prevents the use of object-oriented mechanisms, such as polymorphism and inheritance.

Swiss Army Knife. A Swiss Army Knife corresponds to a complex class that offers many services, for example, a class that implements a large number of interfaces. A Swiss Army Knife is different from a Blob because it exposes high complexity to address all foreseeable needs of a part of a system, while the Blob is a singleton hogging all the processing/control flow of a system.

6.3.4 Data Collection and Oracles

Moha et *al.* [2009] built four oracles of four design smells, for three systems based on the results of DETEX. For each system and each design smell, the detected classes have been checked by independent engineers to assess whether they are true or false positive. The engineers manually validated the detected classes depending on the design smell definition and their context.

We reuse these oracles to assess both the precision and recall of DETEX and SMURF.

For the comparison of BDTEX and SMURF, we used the oracles provided by Khomh et *al.* [2011b] [4]. They manually checked each class of the program to assess whether it is an occurrence of design smell or no.

As expressed in the following equations, we define the precision and the recall.

$$Precision = \frac{NTP}{NTP+NFP}$$
$$Recall = \frac{NTP}{NTP+NFN}$$

where: NTP (number of true positives) is the number of detected occurrences of an design smell that are true occurrences; NFP (number of false positives) is the number of detected occurrences

[4] `http://www.ptidej.net/download/experiments/jss10/`

of an design smell that are false occurrences; NFN (number of false negatives) is the number of occurrences of an design smell that are not detected.

We use Weka[5] to implement SMURF, using its SVM classifier. In all the experiments, we train SMURF on training datasets TDS and apply it on detection datasets DDS.

6.3.5 Analysis Methods

For the purpose of the comparison of DETEX with SMURF, we build for each system and each design smell, three datasets that are composed of two parts: design smells and non-design smell classes in equal numbers. For example, if we consider Blob and ArgoUML, we build three datasets DDS_1, DDS_2, and DDS_3.

To build each dataset we use 30 Blob classes in ArgoUML (identified with the oracles) and add 30 non-Blob classes by choosing them randomly in the remaining classes of ArgoUML. We then divide the 60 classes randomly into the three datasets DDS_1, DDS_2, and DDS_3, making sure that each dataset contains 10 Blob classes and 10 non-Blob classes. To answer our research questions, we perform a 10-fold cross validation that leads us to conduct 264 experiments: four design smells, three systems, and 10 different sets of non-design smell classes. We choose the non design smell classes 10 different times and always consider the average of the 10 runs to avoid any bias due to the random choice of the non design smell classes.

For the comparison of SMURF with BDTEX, we use the same trained and test data used by Khomh et *al.* [2011b].

RQ1. To answer $RQ1_1$ (respectively $RQ1_2$), we train SMURF on a dataset DDS_1 and detect occurrences of an design smell on DDS_2 (respectively on the rest of the whole system). We compute the precision and recall of SMURF on DDS_2 (respectively we compute the number recovered occurrences of BLOB on the rest of the whole system). We then run DETEX on the same dataset, DDS_2 (respectively on the rest of the whole system), and compute its precision and recall.

RQ2. To answer RQ2, we train SMURF and detect occurrences of an design smell on respectively the same trained dataset and test dataset used by Khomh et *al.* [2011b] for the system Xerces for the three design smells they studied. We compute the precision and recall of SMURF on that test dataset. We then compute the precision and recall of BDTEX at different threshold levels according to their results on that test dataset.

[5] `http://www.cs.waikato.ac.nz/ml/weka/`

RQ3. To answer RQ3, we train SMURF on a dataset DDS_1 from one system and detect occurrences of an design smell on DDS_2, from either the same system or another system. We compute the precision and recall of SMURF applied to the DDS_2 of the same system and also that of SMURF applied to the DDS_2 of another system. We then run DETEX on the same dataset DDS_2 of the same/other system, and compute its precision and recall.

RQ4. To answer RQ4, we first train SMURF on DDS_1 and apply it on DDS_2 and compute, as for the other RQs, its precision and recall. We then simulate the practitioners providing their feedback by adding to the trained dataset DDS_1 of SMURF, different percentage (25%, 75% and, 100%) of the content of DDS_3. Indeed, DDS_3 contains a set of labelled design smells and non design smells and then can constitute the set that practitioners would incrementally constitute when validating the detected occurrences as true or false occurrences. For each level of added feedback, we train SMURF on the new trained dataset and re-apply it on DDS_2 and again compute its precision and recall.

6.4 Study Results

We now report the results of our empirical study. These results are discussed in following section. The data for replication purpose and the other results are available online[6].

6.4.1 Accuracy of SMURF Compared to That of DETEX

Figures 6.2 and 6.3 report the precision and recall values when applying DETEX and SMURF on different subsets (DDS_2 dataset). The results of SMURF are better than that of DETEX on all the different subsets. We observe that on small number of classes (25% of classes of the system) DETEX has poor performance because it cannot detect occurrences of design smells without the entire system being present because it uses boxplots and thresholds to identify design smells. On the contrary, SMURF has acceptable precision and recall values. We observe in Figure 6.3 (50% subset) that DETEX has a precision of 100% while SMURF has a precision of around 70% but this is explained by the fact that DETEX detected only 4 classes (out of 38 Spaguetti Code) that are really Spaguetti Code. This leads to 100% of precision. Yet, DETEX missed 34 classes and considered them as safe classes but there are actually Spaguetti Code. This is why with this subset the recall of DETEX is only 10%. It is important to have a balance between precision and recall values. If we increase the precision and drastically decrease the recall, this means that we

[6] http://www.ptidej.net/download/experiments/wcre12a/

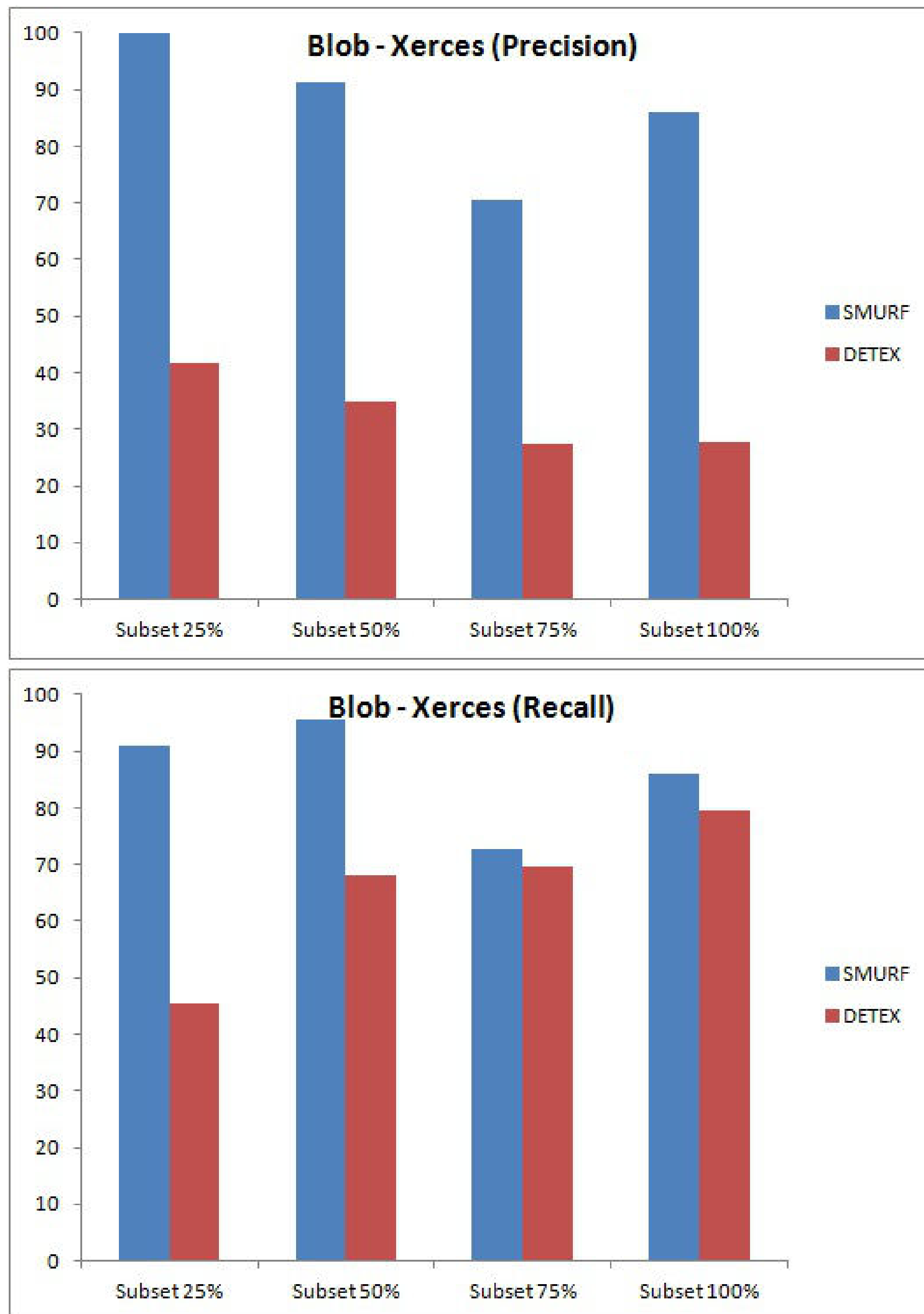


Figure 6.2 – Precision and recall of SMURF and DETEX on different subset size (Blob and Xerces).

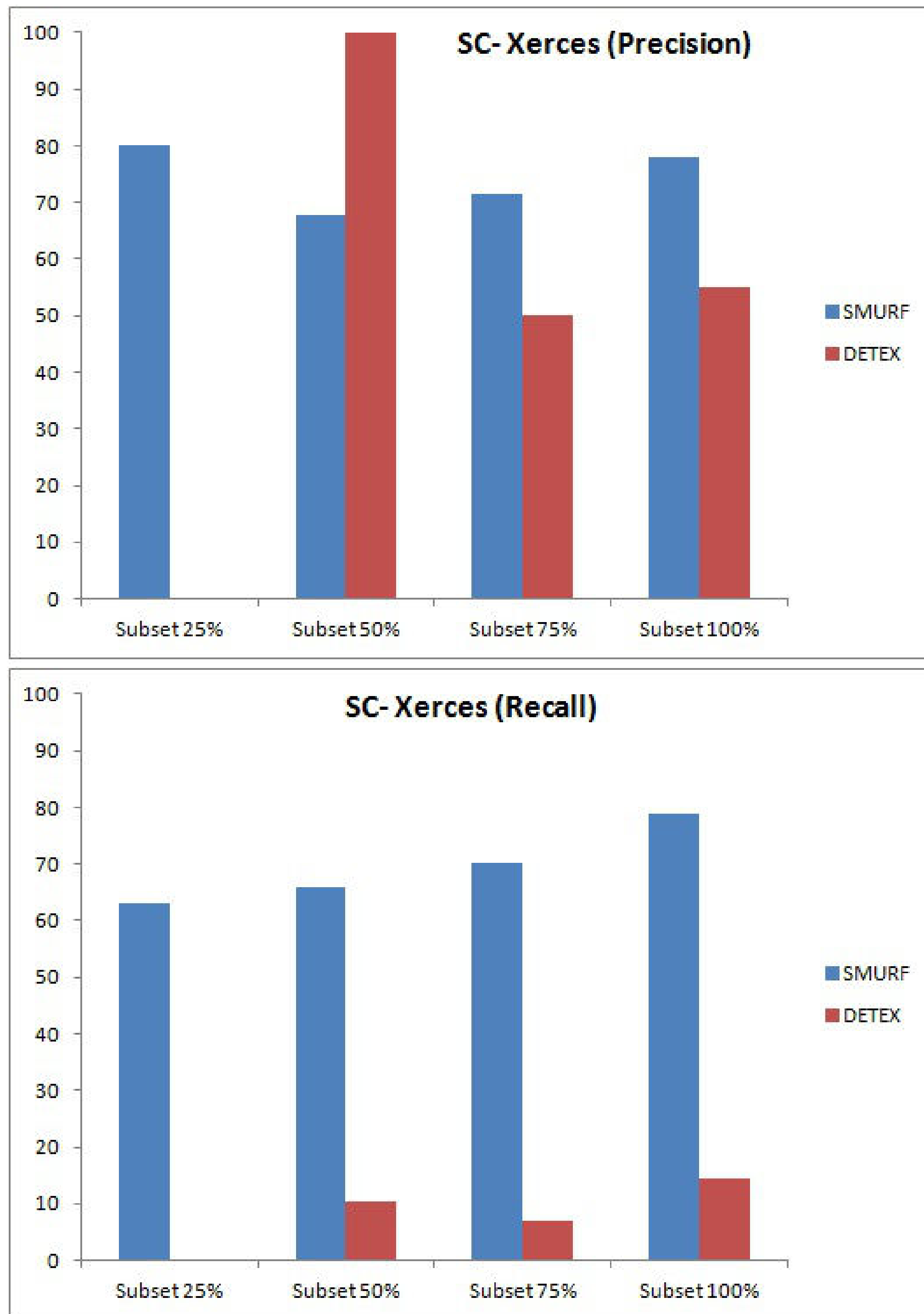


Figure 6.3 – Precision and recall of SMURF and DETEX on different subset size (Spaghetti Code and Xerces).

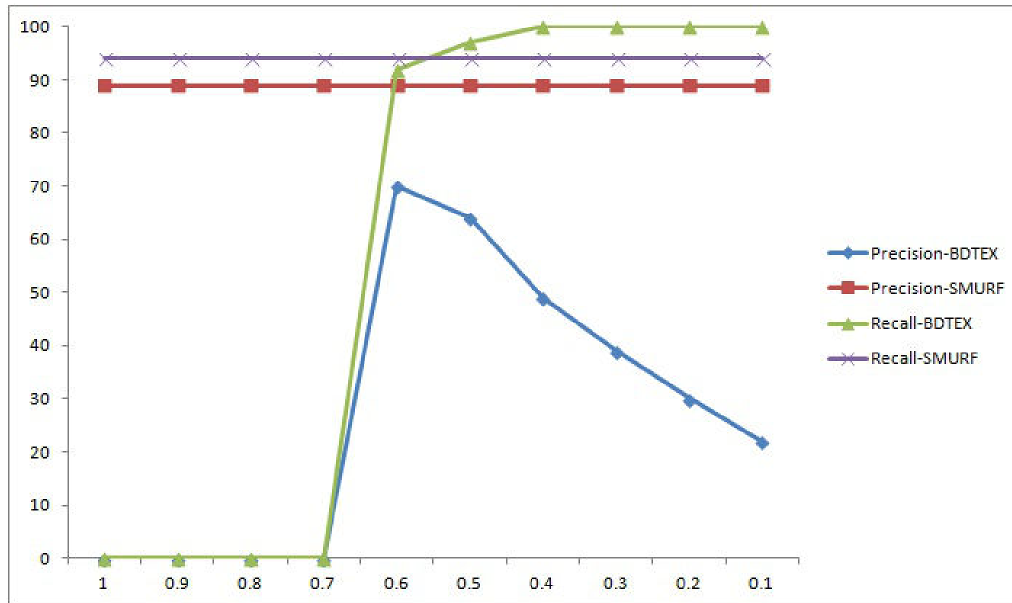


Figure 6.4 – Trends in the increase of precision and recall when decreasing the probabilty of being an design smell for Blob and Xerces.

are missing some occurrences of design smells and we consider them as safe but there are risky classes. If we increase the recall and drastically decrease the precision, this means that we are detecting most of the occurrences of design smells but also report several safe classes as design smells.

> Thus, we answer RQ1: "How does the accuracy of SMURF compare with that of DETEX, in terms of precision and recall?" as follows: on different sizes of subsets of systems, SMURF outperforms DETEX, while on small systems, SMURF can detect occurrences of design smells not detected by DETEX

.

6.4.2 Accuracy of SMURF Compared to That of BDTEX

Figures 6.4 and 6.5 show the accuracy of SMURF and BDTEX when detecting respectively occurrences of Blob and Spaguetti Code in Xerces. It shows that SMURF performs better than BDTEX and is more stable. This observations are discussed in Section 6.5.2

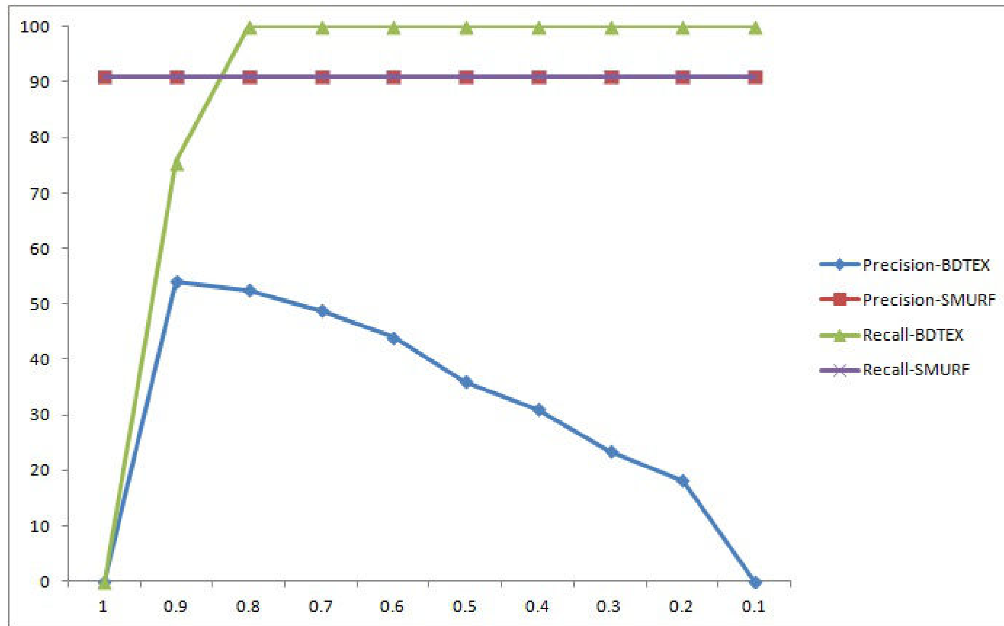


Figure 6.5 – Trends in the increase of precision and recall when decreasing the probabilty of being an design smell for Spaguetti Code and Xerces.

Table 6.2 – Precision of SMURF in inter-systems.

	ArgoUML (%)	Azureus (%)	Xerces (%)
Blob	92.00	96.00	89.00
FD	57.00	62.00	36.00
SC	77.00	74.00	91.00
SAK	56.00	73.00	90.00

Thus, we answer RQ2: "How does the accuracy of SMURF compare with that of BDTEX, in terms of precision and recall when applied on a same entire system?" as follows: SMURF has a better precision and recall than BDTEX.

6.4.3 Accuracy of SMURF in Inter-systems Configurations

Tables 6.2 and 6.3 report the values of precision and recall in the inter-system configuration in which SMURF is trained using the classes of one system (chosen randomly) and applied on the subsets of classes of another system. Evidently, SMURF has quite acceptable values for precision and recall, for inter-system configurations.

Table 6.3 – Recall of SMURF in inter-systems.

	ArgoUML (%)	Azureus (%)	Xerces (%)
Blob	62.00	48.00	94.00
FD	40.00	100.00	20.00
SC	96.00	88.00	91.00
SAK	68.00	84.00	56.00

Thus, we answer RQ3: "How does the accuracy of SMURF change when trained/applied on a same system and trained/applied on different systems, in terms of precision and recall?" as follows: SMURF has a better precision and recall than DETEX. Even in the inter-system configuration, its precision and recall are acceptable in the most of cases excepted for the functional decomposition in the programs ArgoUML (the recall is 40%) and Xerces (the precision is 36% and the recall 20%).

6.4.4 Accuracy of SMURF using Users' Feedback

Figure 6.6 shows the changes in precision and recall values. In future work, we will study other scenario for the feedback (adding only or in different size design smells or non design smells). We observe that the more feedback, the better the precision, up to 100%. For recall, the more feedback, the better the recall but with a slight decrease when we use 100% feedback. We discuss these observations below.

Thus, we answer RQ4: "How does the accuracy of SMURF with relevance feedback compare with that of SMURF without feedback, in terms of precision and recall?" as follows: both precision and recall values increase when taking into account practitioners' feedback.

6.5 Discussion

We now provide detailed discussion on SMURF, results, feedback, and some of our observations.

6.5.1 SMURF vs. DETEX

When applied on different sizes of subsets of systems, we observe that DETEX could not detect occurrences design smells on small subsets and, when it did, the precision and recall values were quite low. We explain this observation by the use of boxplots and thresholds by DETEX. When DETEX analyses a few classes, its use of boxplots and thresholds yield most of the classes to fall

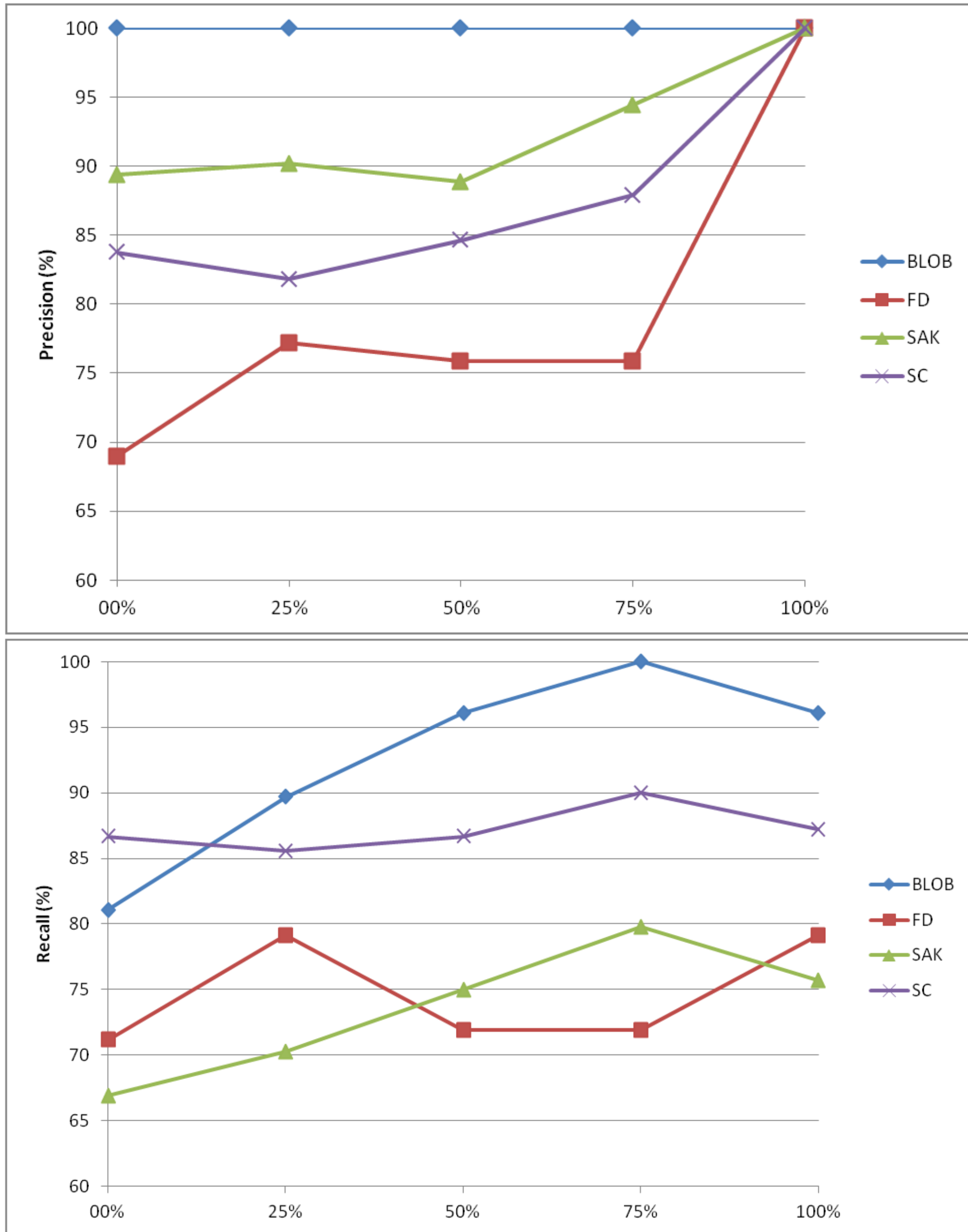


Figure 6.6 – Trends in the increase of precision and recall when integrating incremental feedback.

under (respectively above) the thresholds and hence, not to be reported. The problem does not arise when analysing an entire system because then the boxplots quartiles are different and more classes fall within the threshold values set in the rules. SMURF can work on the whole system and as well as on part of a system. In most of cases, for all configurations (intra or inter-system), SMURF provides acceptable recall and precision values. However we observe that for functional decomposition in the inter-system configuration, the recall is low for ArgoUML(40%) and Xerces (20%) and the precision is low for Xerces (36%). We can explain these results by the fact that, we did not have enough detected occurrences of this design smells. When applying SMURF on the whole system for detecting occurrences of design smell, it performed better than DETEX.

6.5.2 SMURF vs. BDTEX

BDTEX is a probabilistic approach which provides probabilities that a class is an occurrence of design smell. However our SMURF approach is an exact detection approach which is a boolean detection approach telling that a class is an occurrence of an design smell or not. Because the user is not necessarily an expert in the field of design smells, she will refer to a threshold to make her decision to consider a class as an design smell or not. And to compare the precisions and recalls of BDTEX versus SMURF, we consider several decisions thresholds based on probability. We consider that the decision is made to consider a class as an instance of design smell when the probability is 1, then we compute the precision and recall. Then we consider a probability threshold of 0.9 and we compute again the precision and recall. And so on by decreasing probability. The results of this comparison are shown with the figures 6.4 and 6.5 for Blob and Spaguetti Code on Xerces system. The results show that BDTEX contains a high level of uncertainty. For Example, for spaghetti code, while the precision of SMURF is 90%, from a probability threshold of 0.8, the accuracy begins to drop below 50% quickly and significantly. For the Blob while SMURF is 90% precision BDTEX remains at 0% precision even if the probability constraint relaxes up to 0.7. Its from 0.6 probability that the accuracy of BDTEX goes up without reaching the level of that of SMURF and decreases immediately after. The recall is almost zero when at the beginning the requirement is high, but when we decided to release the constraint, it rises to the level of SMURF or even exceed, which is quite normal since in this case BDTEX accepts most classes. Thus, at this level, BDTEX has high recall but bad precision.

6.5.3 Other Design smells/Systems

SMURF requires labelled data to train its SVM. Having this labelled data is a limitation of our approach. However, we claim that it is easier for practitioners to label classes as being occurrences

of some design smells than to write rules and choose right threshold values. Indeed, it is easier for practitioners to recognise an design smell when seeing one rather than to define the rules to detect it [Brown *et al.*, 1998]. This observation relates to the concept of "quality without a name" by Alexander [Alexander *et al.*, 1978], who suggested that it is easy to assess the quality of something when seeing it but difficult to define this very same quality ex nihilo. Thus, we claim that practitioners can easily and incrementally build their own oracles to train SMURF. SMURF would fit in their context and, thanks to the feedback loop in SMURF, practitioners could keep on improving its results as time goes, by adding/removing more labelled data as they find new occurrences of some design smells and decide that an existing occurrence should not be reported any longer or not.

6.5.4 Practitioners' Feedback

SMURF allows practitioners to provide their feedback. The feedback is an easy and valuable way to improve the approach accuracy as confirmed by the results in figure 6.6. In most of the cases, when we increase the feedback provided to SMURF, the recall and the precision increase. However, we observe that when using 100% of the feedback, the recall of SMURF decreases slightly when compared to 75% of the feedback (but is still higher than when not using any feedback). We explain this observation by the fact that our oracles, obtained from a previous work [Moha *et al.*, 2009], may not be completely accurate and, thus, integrating their data could lead to misclassification of some true positive occurrences. Future work includes revising entirely the oracles to identify misclassified occurrences, if any.

6.5.5 Software Evolution

With previous approaches [Moha *et al.*, 2009], practitioners needed to run the whole design smell detection process every now and then. Re-executing the detection process meant that practitioners must then manually verify all the reported occurrences again the possible occurrences that have already been reported in a previous run. SMURF overcomes this problem by using the previously set of manually-validated design smells as oracle, either for training or as feedback. For example, if a practitioner built an design smells' oracle AP_{oracle} for ArgoUML version 1. SMURF can use AP_{oracle} to train its SVM to detect design smells in the subsequent versions of ArgoUML, which lessens the limitation of SMURF concerning labelled data. Thus, as a software system evolves, the related training/feedback data will also increase in size and yield an improvement in the accuracy of SMURF and reduction in the manual validation.

6.5.6 SMURF and Practitioners

Previous approaches depended heavily on rules and thresholds. To define rules, a practitioner must have a detailed technical knowledge of design smells and the underlying framework. There is no general rule to define what an design smell is. Thus, it is difficult for a practitioner to write a rule. For example, it is quite possible for a practitioner to state that a Blob class must contain 300 LOC and for another practitioner this could be $30,000$ LOC. Defining a wrong threshold and–or rule would negatively impact the results of the approach and would make it useless for the practitioners. SMURF overcomes both of these limitations, *i.e.*, use of design smell rules and thresholds. SMURF can detect occurrences of design smells in object-oriented systems without having to manually set rules for detection. Also, SMURF has a good precision and recall, even in the inter-system configuration. Finally, SMURF provide better results to practitioners as they integrate their feedback.

6.6 Threats to Validity

We now discuss threats to the validity of our results [Yin, 2008].

Internal Validity. In our study, threats to internal validity are mainly related to the identification of classes suspected to be occurrences of design smells. To reduce the effect of this threat, we used previous oracles of occurrences of design smells reported by Moha et al [Moha *et al.*, 2009]. These occurrences have been manually validated by independent engineers.Concerning the potential dependence of the obtained results on the chosen design smells and systems, our study is not affected. Indeed, we used four well-known and representative design smells. These design smells also have been used in previous works. Further, we applied our approach to three open-source systems with different size and these systems have also been used by previous researchers. Moreover, we limited the bias of intrinsic randomness of our results by repeating each experiment 10 times for each design smell per system; a total of 264 experiments before and after feedback to compare the results.

Reliability Validity. Reliability validity threats concern the possibility of replicating the study concerned. To mitigate this threat, we used open-source systems that can be freely downloaded from the Internet. We attempted to provide all the necessary details to replicate our study. Moreover, the results of the validation and the datasets are available online.

External Validity. Threats to external validity concern the possibility to generalise our results. We studied three systems with different sizes and different domains. In spit of the fact that three systems might not be a very large number that can be generalized, as the systems were of varying sizes, and domains, we can claim that the results can be generalized with certain degree of confidence. Further, we also used a representative subset of design smells. However, we will apply SMURF on other systems and design smells in future work to negate the threat completely.

6.7 Summary

Design smells are a fact of developers' life when developing software systems under the conditions prevailing nowadays: distribution in time and space, time pressure, complexity, agile software development context. In particular, design smells impede program comprehension [Abbes *et al.*, 2011] and thus have negative impact on both development and changes. We observed that current design smell detection approaches had four limitations: (1) they require extensive knowledge of design smells, (2) they have limited precision and recall, (3) they are not incremental and iterative, and (4) they cannot be applied on subsets of systems.

To overcome these limitations, we introduced a novel approach to detect design smells, SMURF, based on support vector machines (SVM). SUMRF is an iterative and incremental detection approach that allows (1) learning the detection rules from a set of known occurrences of design smells and non design smells, and (2) improving accuracy. Thus, SMURF, overcomes the previous limitations. Practitioners can easily produce the needed set according to their needs and context.

We designed an empirical study that allowed us to compare the results of DETEX [Moha *et al.*, 2009] and BDTEX [Khomh *et al.*, 2011b], the best two state of the art approaches, respectively, in exact and probabilistic design smells detections, with the results of SMURF. We performed more than 300 experiments to show how SMURF performs on a set of three programs (ArgoUML v0.19.8, Azureus v2.3.0.6, and Xerces v2.7.0) using four design smells (Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife).

We showed that the accuracy of SMURF is greater than that of DETEX and BDTEX when detecting design smells on a set of classes or on the whole system. We also showed that SMURF can be applied in both intra-system and inter-system configurations. Finally, we reported that SMURF accuracy improves when using practitioners' feedback.

We thus conclude that our conjecture is correct: SVM-based approach can overcome the four limitations of previous approaches and could be more readily adopted by practitioners.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the results and conclusions of our thesis. We also discuss opportunities for future work.

Contents

7.1 Conclusions **94**
7.2 Future Work **96**

7.1 Conclusions

Software maintenance and evolution are important in software development. Therefore, in software maintenance and evolution comprehension and change are central. Then, it become crucial to analyze the factors that could influence comprehesion and change. Therefore, in this thesis, we analyzed the impact of design smells. We conducted an empirical study to bring evidence of the impact of design smells on comprehension. We designed and conducted two experiments with 59 subjects to assess the impact of the composition of two Blob or two Spaghetti Code, on the performance of developers performing comprehension tasks. We measured developers' performance using: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. The results of the two experiments showed that two occurrences of Blob or Spaghetti Code design smells impedes significantly developers performance during comprehension and change. The obtained results justify a posteriori previous researches on the specification and detection of design smells. Software development teams should warn developers against high number of occurrences of design smells and recommend refactorings at each step of the development to remove them when possible.

Then, we investigated the relation between design smells and faults in classes from the point of view of developers who must fix faults. We studied the impact of the presence of design smells on the effort required to fix faults, which we measure using three metrics: (1) the duration of the fixing period; (2) the number of fields and methods impacted by fault-fixes; and, (3) the entropy of the fault-fixes in the source code. We conducted an empirical study with 12 design smells detected in 54 releases of four systems: ArgoUML, Eclipse, Mylyn, and Rhino. Our results showed that the duration of the fixing period is longer for faults involving classes with design smells. Also, fixing faults in classes with design smells impacts more files, more fields, and more methods. We also observed that after a fault is fixed, the number of occurrences of design smells in the classes involved in the fault decreases. Understanding the impact of design smells on development effort is important to help development teams better assess and forecast the impact of their design decisions and therefore lead their effort to improve the quality of their software systems. Development teams should monitor and remove design smells from their software systems because they are likely to increase maintenance efforts.

Finaly, we analysed the problem of design smells detection. In the context of software maintenance and evolution it is important to have a tool to detect design smells incrementally and iteratively. This incremental and iterative detection process could reduce costs, effort, and resources by allowing practitioners to identify and take into account occurrences of design smells as they find them during comprehension and change. Researchers have proposed approaches to detect occurrences of design smells but these approaches have currently four limitations: (1) they require extensive knowledge of anti-patterns; (2) they have limited precision and recall; (3) they are not incremental; and (4) they cannot be applied on subsets of systems. To overcome these limitations and propose an incremental and iterative tool, we introduced SMURF, a novel approach to detect design smells, based on a machine learning technique—support vector machines—and taking into account practitioners' feedback. Indeed, through an empirical study involving three systems and four design smells, we showed that the accuracy of SMURF is greater than that of DETEX and BDTEX (the state of the art) when detecting design smells occurrences. We also showed that SMURF can be applied in both intra-system and inter-system configurations. Finally, we reported that SMURF accuracy improves when using practitioners' feedback.

We thus, confirm our thesis: we bring quantitative evidence of the impact of design smells on comprehension and fault-fixing effort and provide a tool for accurate incremental and iterative design smells detection.

7.2 Future Work

To strengthen the evidence of the impact of design smells on comprehension, future work includes investigating the relation between the number of occurrences of design smells and understandability (*i.e.*, subjects' efforts, times, and percentages of correct answers). For example, we plan to investigate the impact of three and more occurrences of a design smells on systems understandability. We also plan to replicates the studies in other contexts, with other subjects, other questions, other design smells, and other systems.

For the impact of design smells on fault and fault-fixing effort, we plan to replicate our studies with other software systems and other design smells. Other than using only open-source projects, we plan to extend this study to industrial systems. Last, but not least, a replication of this study on design patterns is desirable to compare the impact of design smells on fault fixing effort with the impact of design patterns on fault fixing effort.

For incremental an iterative design smells detection tools, future work includes performing an empirical study about the use of SMURF in real-world environments. We will ask our industrial partners help in realising such a study. Further, we would also reproduce the study with other systems and design smells to increase our confidence in the generalisability of our conclusions. Another study could be the evaluation of the impact of the quality of feedback on SMURF results.

Future work includes also the study on how long does a design smell survive. We will study the factors of introduction and propagation of design smells in software systems. We will investigate the factors of extinction of design smells and design a tool to propose refactorings for design smells

Bibliography

[Abbes *et al.*, 2011] cited pages 3, 5, 22, 29, 30, 37, 39, 40, 47, 74, 93

Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In Tom Mens, Yiannis Kanellopoulos, and Andreas Winter, editors, *CSMR, 15th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE Computer Society, 2011.

[Alexander *et al.*, 1978] cited page 90

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1st edition, August 1978. ISBN: 0-19-501919-9.

[Alikacem and Sahraoui, 2006] cited pages 6, 74

El Hachemi Alikacem and Houari A. Sahraoui. Détection d'anomalies utilisant un langage de règle de qualité. In *LMO*, pages 185–200. Hermes Science Publications, 2006.

[Ballis *et al.*, 2008a] cited pages 24, 25

D. Ballis, A. Baruzzo, and M. Comini. A rule-based method to match software patterns against uml models. *Electron. Notes Theor. Comput. Sci.*, 219:51–66, Amsterdam, The Netherlands, The Netherlands, November 2008. Elsevier Science Publishers B. V.

[Ballis *et al.*, 2008b] cited pages 24, 25

Demis Ballis, Andrea Baruzzo, and Marco Comini. A minimalist visual notation for design patterns and antipatterns. In *Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 51–56, Washington, DC, USA, 2008. IEEE Computer Society. ISBN: 978-0-7695-3099-4.

[Basu *et al.*, 2008] cited page 26

Sugato Basu, Ian Davidson, and Kiri Wagstaff. *Constrained Clustering: Advances in Algo-*

rithms, Theory, and Applications. Chapman & Hall/CRC, 1 edition, 2008. ISBN: 1584889969, 9781584889960.

[Beck *et al.*, 2001] cited page 2
Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[Bedo *et al.*, 2006] cited pages 25, 75
Justin Bedo, Conrad Sanderson, and Adam Kowalczyk. An efficient alternative to svm based recursive feature elimination with applications in natural language processing and bioinformatics. In Abdul Sattar and Byeong-ho Kang, editors, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 170–180. Springer Berlin Heidelberg, 2006.

[Bennett and Rajlich, 2000] cited page 1
Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM. ISBN: 1-58113-253-0.

[Bois *et al.*, 2006] cited pages 22, 30
Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering'06*, pages 346–355, 2006.

[Brown *et al.*, 1998] cited pages 2, 9, 21, 28, 42, 74, 90
William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998. ISBN: 0-471-19713-0.

[Brown, 1996] cited page 80
Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical report TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1996.

[Cao *et al.*, 2008] cited page 25
Guihong Cao, Jian-Yun Nie, Jianfeng Gao, and Stephen Robertson. Selecting good expansion terms for pseudo-relevance feedback. In *Proceedings of the 31st Annual International*

ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore, July 20-24, 2008, pages 243–250. ACM, 2008. ISBN: 978-1-60558-164-4.

[Cepeda Porras and Guéhéneuc, 2010] cited page 41
Gerardo Cepeda Porras and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of different design pattern representations in uml class diagrams. *Empirical Softw. Engg.*, 15(5):493–522, Hingham, MA, USA, 2010. Kluwer Academic Publishers.

[Chatzigeorgiou and Manakos, 2010] cited page 23
Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, QUATIC '10, pages 106–115, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-0-7695-4241-6.

[Choi *et al.*, 2012] cited pages 25, 75
Myung Jin Choi, Antonio Torralba, and Alan S. Willsky. A tree-based context model for object recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34:240–252, Washington, DC, USA, February 2012. IEEE Computer Society.

[Coplien and Harrison, 2005] cited page 28
James O. Coplien and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Upper Saddle River, NJ (2005), 1st edition, 2005.

[Cortes and Vapnik, 1995] cited page 16
Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297. Springer Netherlands, 1995. 10.1007/BF00994018.

[D'Ambros *et al.*, 2010] cited page 44
Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *QSIC '10: Proceedings of the 2010 10th International Conference on Quality Software*, pages 23–31, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-0-7695-4131-0.

[Duchowski, 2007] cited page 35
Andrew T. Duchowski. *Eye Tracking Methodology: Theory and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN: 1846286085.

[Eaddy *et al.*, 2008] cited pages 14, 69
Marc Eaddy, Thomas Zimmermann, Kaitin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Software Eng.*, 34(4):497–515, 2008.

[Fischer *et al.*, 2003] cited page 14

Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *In Proceedings of the International Conference on Software Maintenance*, pages 23–32, 2003.

[Fowler, 1999a] cited pages 2, 9, 21, 74

Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999. ISBN: 0-201-48567-2.

[Fowler, 1999b] cited page 32

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[Gamma *et al.*, 1994] cited page 2

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Grissom and Kim, 2005] cited page 20

R.J. Grissom and J.J. Kim. *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates, 2005. ISBN: 9780805850147.

[Guéhéneuc and Antoniol, 2008] cited page 76

Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multi-layered framework for design pattern identification. In Sebastian Elbaum and David S. Rosenblum, editors, *Transactions on Software Engineering (TSE)*, 34(5):667–684. IEEE Computer Society Press, September 2008. 18 pages.

[Guéhéneuc *et al.*, 2004] cited page 76

Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *Proceedings of the* 11th *Working Conference on Reverse Engineering (WCRE)*, pages 172–181. IEEE Computer Society Press, November 2004. 10 pages.

[Hart and Stavenland, 1988] cited page 34

S. G. Hart and L. E. Stavenland. In P. A. Hancock and N. Meshkati, editors, Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. pages 139–183. Elsevier, 1988.

[IEEE, 1999] cited page 1

IEEE. *IEEE Standard for Software Maintenance, IEEE Std 1219-1998*, volume 2. IEEE Press, 1999.

[Ignatios *et al.*, 2003] cited pages 2, 22

Deligiannis Ignatios, Stamelos Ioannis, Angelis Lefteris, Roumeliotis Manos, and Shepperd Martin. A controlled experiment investigation of an object oriented design heuristic for maintainability. In Elsevier, editor, *Journal of Systems and Software*, 65(2). Elsevier, February 2003.

[Ignatios *et al.*, 2004] cited pages 2, 22

Deligiannis Ignatios, Shepperd Martin, Roumeliotis Manos, and Stamelos Ioannis. An empirical investigation of an object-oriented design heuristic for maintainability. In Elsevier, editor, *Journal of Systems and Software*, 72(2). Elsevier, 2004.

[John Shawe-Taylor, 2000] cited page 18

Nello Cristianini John Shawe-Taylor. Support vector machines and other kernel-based learning methods. *Cambridge University Press*, 2000.

[Kessentini *et al.*, 2010] cited page 24

Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 113–122, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0116-9.

[Khomh and Guéhéneuc, 2008] cited pages 2, 29

Foutse Khomh and Yann-Gaël Guéhéneuc. Do design patterns impact software quality positively? In Christos Tjortjis and Andreas Winter, editors, *Proceedings of the* 12th *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, April 2008.

[Khomh *et al.*, 2009a] cited page 30

Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the* 16th *Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, October 2009.

[Khomh *et al.*, 2009b] cited pages 10, 13

Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *WCRE*, pages 75–84, 2009.

[Khomh *et al.*, 2011a] cited pages 2, 3, 5, 10, 13, 14, 47, 48, 49, 55, 71, 74

Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Journal of Empirical Software Engineering (EMSE)*, 2011.

[Khomh *et al.*, 2011b] cited pages 13, 25, 26, 75, 78, 80, 81, 82, 93

Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *J. Syst. Softw.*, 84(4):559–572, New York, NY, USA, April 2011. Elsevier Science Inc.

[Khomh *et al.*, 2012] cited page 23

Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[Kim *et al.*, 2008] cited page 25

Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[Langelier *et al.*, 2005] cited page 24

Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 214–223, New York, NY, USA, 2005. ACM. ISBN: 1-58113-993-4.

[Lucia *et al.*, 2012] cited page 25

Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. Active refinement of clone anomaly reports. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 397–407, Piscataway, NJ, USA, 2012. IEEE Press. ISBN: 978-1-4673-1067-3.

[Mäntylä, 2003] cited page 22

Mika Mäntylä. *Bad Smells in Software - a Taxonomy and an Empirical Study.* Ph.D. thesis, Helsinki University of Technology, 2003.

[Marinescu, 2004a] cited pages 6, 23

Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20th *International Conference on Software Maintenance*, pages 350–359. IEEE CS Press, 2004.

[Marinescu, 2004b] cited pages 24, 25, 26, 74

Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *In Proceedings of the IEEE 20th International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[Martin, 2003] cited page 2
Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN: 0135974445.

[Moha and Guéhéneuc, 2005] cited pages 30, 32
Naouel Moha and Yann-Gaël Guéhéneuc. On the automatic detection and correction of software architectural defects in object-oriented designs. In *In Proceedings of the 6 th ECOOP Workshop on Object-Oriented Reengineering. Universities of Glasgow and Strathclyde*, 2005.

[Moha *et al.*, 2009] cited pages 2, 3, 6, 10, 13, 23, 24, 25, 26, 74, 75, 78, 79, 80, 81, 91, 92, 93
Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. In Mark Harman, editor, *Transactions on Software Engineering (TSE)*. IEEE Computer Society Press, 2009.

[Moha *et al.*, 2010a] cited pages 30, 32, 69
Naouel Moha, Yann G. Guéhéneuc, Laurence Duchien, and Anne F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[Moha *et al.*, 2010b] cited page 13
Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[Moha *et al.*, 2010c] cited pages 2, 3, 47
Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Asp. Comput.*, 22(3-4):345–361, 2010.

[Munro, 2005] cited page 23
Matthew James Munro. Product metrics for automatic identification of “bad smell” design problems in java source-code. In *Proceedings of the* 11th *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[Olbrich *et al.*, 2009] cited page 22
Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Third International Symposium on Empirical Software Engineering and Measurement*, 2009.

[Oliveto *et al.*, 2010] cited page 23

Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In Rudolf Ferenc Rafael Capilla and Juan Carlos Dueas, editors, *Proceedings of the* 14th *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010.

[Onoda *et al.*, 2006] cited pages 25, 26

Takashi Onoda, Hiroshi Murata, and Seiji Yamada. Non-relevance feedback document retrieval based on one class svm and svdd. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN 2006, part of the IEEE World Congress on Computational Intelligence, WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006*, pages 1212–1219, 2006.

[Parnas, 1994] cited page 29

David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN: 0-8186-5855-X.

[Pressman, 2001] cited page 1

Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, November 2001. ISBN: 0-07-249668-1.

[PressmanR.S., 1996] cited page 47

PressmanR.S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 1996. ISBN: 0070521824.

[Rahma Fourati and Abdallah, 2011] cited pages 24, 25, 26

Nadia Bouassida Rahma Fourati and Hanne Ben Abdallah. A metric-based approach for anti-pattern detection in uml designs. In *COMPUTER AND INFORMATION SCIENCE 2011*, CISW '07, 2011.

[Riel, 1996] cited pages 21, 80

Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Setia *et al.*, 2005] cited pages 25, 26

Lokesh Setia, Julia Ick, and Hans Burkhardt. Svm-based relevance feedback in image retrieval using invariant feature histograms, 2005.

[Settas *et al.*, 2012] cited page 24

Dimitrios Settas, Antonio Cerone, and Stefan Fenz. Enhancing ontology-based antipattern detection using bayesian networks. *Expert Syst. Appl.*, 39(10):9041–9053, Tarrytown, NY, USA, August 2012. Pergamon Press, Inc.

[Sillito, 2007] cited pages 29, 35

Jonathan Sillito. *Asking and answering questions during a programming change task*. Ph.D. thesis, Vancouver, BC, Canada, Canada, 2007. ISBN: 978-0-494-26794-3.

[Foutse Khomh *et al.*, 2009] cited page 23

Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the* 9th *International Conference on Quality Software (QSIC)*. IEEE CS Press, August 2009. 10 pages.

[Travassos *et al.*, 1999] cited page 23

Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the* 14th *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–56. ACM Press, 1999.

[Vaucher *et al.*, 2009] cited page 30

Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Tracking design smells: Lessons from a study of god classes. *Reverse Engineering, Working Conference on*, 0:145–154, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[von Mayrhauser and Vans, 1995] cited pages 4, 28

Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[Wake, 2003] cited page 22

William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN: 0321109295.

[Webster, 1995] cited pages 2, 21

Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1st edition, February 1995. ISBN: 1558513973.

[Wirfs-Brock and McKean, 2002]

Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley Professional, 2002. ISBN: 0201379430.

[Wohlin *et al.*, 2000] cited pages 30, 34, 35, 41

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN: 0-7923-8682-5.

[Yamashita and Moonen, 2012] cited page 23

Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, sept. 2012.

[Ye *et al.*, 2011] cited pages 25, 75

Zheng Ye, Jimmy Xiangji Huang, and Hongfei Lin. Incorporating rich features to boost information retrieval performance: A svm-regression based re-ranking approach. *Expert Syst. Appl.*, 38:7569–7574, Tarrytown, NY, USA, June 2011. Pergamon Press, Inc.

[Yin, 2008] cited pages 69, 92

Robert K Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, Inc, 2008.

[Zaman *et al.*, 2011] cited pages 14, 15

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102. IEEE Computer Society, 2011.

[Zhou *et al.*, 2007] cited page 26

Yihua Zhou, Weimin Shi, Lijuan Duan, and Cuiying Niu. A relevance feedback algorithm based on svm model's dynamic adjusting for image retrieval. In *Proceedings of the 2007 International Conference on Computational Intelligence and Security Workshops*, CISW '07, pages 287–290, Washington, DC, USA, 2007. IEEE Computer Society. ISBN: 0-7695-3073-7.

Appendix A

Definitions of Metrics

This Appendix presents the definitions of all the metrics used in this thesis.

ACAIC: ancestor Class-Attribute Import Coupling.

ACMIC: ancestors Class-Method Import Coupling.

AID: average Inheritance Depth of an entity.

ANA: count the average number of classes from which a class inherits informations.

CAM: computes the relatedness among methods of the class based upon the parameter list of the methods.

CBOin: coupling Between Objects of one entity.

CBOout: coupling Between Objects of one entity.

CIS: counts the number of public methods in a class.

CLD: class to Leaf Depth of an entity.

cohesionAttributes: returns the degree of cohesion between methods and attributes of a class.

connectivity: returns the degree of connectivity of an entity in a system.

CP: the number of packages that depend on the package containing entity.

DAM: returns the ratio of the number of private (protected) Attributes to the total number of Attributes declared in a class.

DCAEC: returns the DCAEC (Descendants Class-Attribute Export Coupling) of one entity.

DCC: returns the number of classes a class is directly related to (by attribute declarations and message passing.

DCMEC: returns the DCMEC (Descendants Class-Method Export Coupling) of one entity.

DIT: returns the DIT (Depth of inheritance tree) of an entity.

DSC: count of the total number of classes in the design.

EIC: the number of inheritance relationships in which superclasses are in external packages.

EIP: the number of inheritance relationships where the superclass is in the package containing entity and the subclass is in another package.

ICHClass: compute the complexity of an entity as the sum of the complexities of its declared and inherited methods.

LCOM1: returns the LCOM (Lack of COhesion in Methods) of an entity.

LCOM2: returns the LCOM (Lack of COhesion in Methods) of an entity.

LOC: returns the number of line of code of an entity.

MFA: the ratio of the number of methods inherited by a class to the number of methods accessible by member methods of the class.

MOA: count the number of data declarations whose types are user defined classes.

NAD: number of attributes declared.

NADExtended: number of attributes declared in a class and in its member classes.

NCM: returns the NCM (Number of Changed Methods) of an entity.

NCP: the number of classes package containing entity.

NMA: returns the NMA (Number of New Methods) of an entity.

NMD: number of methods declared.

NMDExtended: number of methods declared in the class and in its member classes.

NMI: returns the NMI (Number of Methods Inherited) of an entity.

NMO: returns the NMO (Number of Methods Overridden) of an entity.

NOA: returns the NOA (Number Of Ancestors) of an entity.

NOC: returns the NOC (Number Of Children) of an entity.

NOD: returns the NOD (Number Of Descendents) of an entity.

NOH: count the number of class hierarchies in the design.

NOM: counts all methods defined in a class.

NOP: returns the NOP (Number Of Parents) of an entity.

NOParam: compute the average number of parameters of methods.

NOPM: count of the Methods that can exhibit polymorphic behavior.

PIIR: the number of inheritance relationships existing between classes in the package containing entity.

PP: the number of provider packages of the package containing entity.

REIP: EIP divided by the sum of PIIR and EIP.

RFP: the number of class references from classes belonging to other packages to classes belonging to the package containing entity.

RPII: PIIR divided by the sum of PIIR and EIP.

RRFP: RFP divided by the sum of RFP and the number of internal class references.

RRTP: RTP divided by the sum of RTP and the number of internal class references.

RTP: the number of class references from classes in the package containing entity to classes in other packages.

SIX: returns the SIX (Specialisation IndeX) of an entity.

WMC1: computes the weight of an entity considering the complexity of a method to be unity.

McCabe: number of points of decision + 1.

CBO: coupling Between Objects of one entity.

LCOM5: returns the LCOM (Lack of COhesion in Methods) of an entity.

WMC: computes the weight of an entity by computing the number of method invocations in each method.

PageRank: measures the relative importance of a class in the overall structure of relations among classes.

Appendix B

Specification of Code Smells and Antipatterns

This Appendix presents the definitions of code smells and antipatterns studied in this thesis.

B.1 Detailed Definitions of the code Smells

In this thesis we focused on the following code smells:

AbstractClass: this code smell is characteristic of the Speculative Generality Antipattern. This odor exists when we have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future.

ChildClass: this code smell occurs when the number of methods declared in a class and the number of it's declared attributes is very high. It is a symptom of poor object decomposition. The public interface of the class differing greatly from the one of its super-class. This code smell characterises the Tradition Breaker antippatern.

ClassGlobalVariable: this code smell occurs when a class declares public class variable that are used as "global variable" in procedural programming.

ClassOneMethod: this code smell occurs when a class has only one method.

ComplexClassOnly: this code smell is present when a class both declares many fields and methods and which methods realise complex treatments, using many if and switch instructions.

Such a class is probably providing lots of services while being difficult to maintain and fragile due to its complexity.

ControllerClass: this odor is present when a class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes.

DataClass: this code smell is present when a class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

FewMethod: this code smell characterise Lazy classes that declare few methods.

FieldPrivate: this code smell is present when many private fields are declared. It's generally symptomatic of the Functional Decomposition antipattern.

FieldPublic: this code smell is symptomatic of the Class Data Should Be Private antippatern. It occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class.

FunctionClass: this code smell occurs when we have a main class, i.e., a class with a procedural name, such as Compute or Display. It is symptomatic of the Functional Decomposition antipattern.

HasChildren: this code smell describes classes with many children.

LargeClass: this odor concerns classes that are trying to do too much. These classes do not follow the good practice of divide-and-conquer which consists of decomposing a complex problem into smaller problems. These classes also have low cohesion.

LargeClassOnly: this code smell concerns classes with a very high number of attributes and/or methods defined.

LongMethod: this odor is a method with a high number of lines of code. A lot of variables and parameters are used. Generally, this kind of method does more than its name suggests it.

LongParameterListClass: this odor corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters. Long lists of parameters in a method, though common in procedural code, are difficult to understand and likely to be volatile.

LowCohesionOnly: this code smell characterises the lack of cohesion in a class.

ManyAttributes: this code smell occurs when the number of attributes declared in a class is too high.

MessageChainsClass: this code smell is present when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects.

MethodNoParameter: this code smell occurs when a class declares methods with no parameter.

MultipleInterface: this code smell occurs when a class implements a high number of interfaces. It is generally symptomatic of the Swiss Army Knife antipattern.

NoInheritance: this odor is present when inheritance is scarcely used.

NoPolymorphism: this odor is present when polymorphism is scarcely used.

NotAbstract: this odor occurs when a developer haven't yet seen how a higher-level abstraction can clarify or simplify his code.

NotClassGlobalVariable: this odor manifest itself in the anipattern Anti-Singleton when a class declares public class variable that are used as "global variable" in procedural programming. It reveals procedural thinking in object-oriented programming and may increase the difficulty to maintain the program.

NotComplex: this code smell characterises classes performing "atomic" functionality, with little complexity.

OneChildClass: this code smell occurs when a class does not have child class.

ParentClassProvidesProtected: this code smell occurs when a subclass does not use attributes and/or methods protected inherited by a parent.

RareOverriding: this code smell occurs when a class rarely overrides inherited attributes and/or methods.

TwoInheritance: this odor characterises a hierarchy with a depth greater than two.

B.2 Detailed Definitions of the Antipatterns

This thesis focused on the following antipatterns:

Anti-Singleton: it is a class that declares public class variable that are used as "global variable" in procedural programming. It reveals procedural thinking in object-oriented programming and may increase the difficulty to maintain the program.

Blob: (called also God class [Riel, 1996]) corresponds to a large controller class that depends on data stored in surrounded data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes [Wirfs-Brock and McKean, 2002].

Class Data Should Be Private: it occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class.

Complex Class: it is a class that both declares many fields and methods and which methods realise complex treatments, using many if and switch instructions. Such a class is probably providing lots of services while being difficult to maintain and fragile due to its complexity.

Large Class: it is a class with too many responsibilities. This kind of class declares a high number of usually unrelated methods and attributes.

Lazy Class: it is a class that does not do enough. The few methods declared by this class have a low complexity.

Long Method: it is a method with a high number of lines of code. A lot of variables and parameters are used.Generally, this kind of method does more than its name suggests it.

Long Parameter List: it corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters.

MessageChains: it Occurs when you have a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects [Fowler, 1999a].

Speculative Generality: it is an abstract class without child classes. It was added in the system for future uses and this entity pollutes the system unnecessarily.

Swiss Army Knife: it refers to a tool fulfilling a wide range of needs. The Swiss Army Knife design smell is a complex class that offers a high number of services, for example, a complex class implementing a high number of interfaces. A Swiss Army Knife is different from a Blob, because it exposes a high complexity to address all foreseeable needs of a part of a system, whereas the Blob is a singleton monopolising all processing and data of a system. Thus, several Swiss Army Knives may exist in a system, for example utility classes.

The Refused Parent Bequest: it appears when a subclass does not use attributes and/or methods public and/or protected inherited by a parent. Typically, this means that the class hierarchy is wrong or badly organized.

The Spaghetti Code: it is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables for processing. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism and inheritance.

Appendix C

Example of Comprehension Questions

Question 1: Where is, in this project, the method involved in the implementation of indicating the health status of the resource to be downloaded, by calculating the number of seeds and peers?

Question 2: What is the type (class) that implements all the services related to the download of a resource, like calculating the connection to any peer or seeding, determine the health status, manage torrent files, etc?

Question 3: Where is the type DownloadManagerStateImpl referenced?

Question 4: What data can we access from an object of the type DownloadManagerImpl?

Question 5: How are the types DownloadManagerImpl and DownloadManagerStateImpl related?

Question 6: What is the behaviour that DownloadManagerStateImpl and DownloadManagerImpl provide together? (Tick two answers)

1. Add a new peer using the given address and port.
2. Calculate the number of peers and seeds
3. Calculate the number of remote connections
4. Download torrents

Appendix D

TLX Rating Scale Definition

Title	Endpoints	Descriptions
MENTAL DEMAND	*Low/High*	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?
PHYSICAL DEMAND	*Low/High*	How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
TEMPORAL DEMAND	*Low/High*	How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?
EFFORT	*Low/High*	How hard did you have to work (mentally and physically) to accomplish your level of performance?
PERFORMANCE	*Good/Poor*	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?
FRUSTRATION LEVEL	*Low/High*	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

Appendix E

Post-mortem Questionnaire

Question 1: Are you familiar with the term “empirical study”? If yes, please give a brief definition.

Question 2: Are you familiar with the term “anti-pattern”? If yes, please give a brief definition. If possible, please provide an example of an anti-pattern.

Question 3: Do you know the definition of “refactoring”? Please give a brief definition.

Question 4: How would you rate your skills and knowledge of software engineering? (Check the box that best describes your level)

bad☐ quiet good☐ good☐ excellent☐ expert☐

Question 5: How would you rate your level of experience in Java?

bad☐ quiet good☐ good☐ excellent☐ expert☐

Question 6: How would you rate your level of experience in Eclipse?

bad☐ quiet good☐ good☐ excellent☐ expert☐

Question 7: How did you find the information provided in the procedure? (You can tick several boxes)

Not too much ☐ correct ☐ too much ☐ simple☐ complex ☐

Question 8: What was the impact of the amount of information provided during the procedure on your performance in accomplishing the requested tasks? (Negative? Positive? Required too much concentration? Sufficient? etc.)

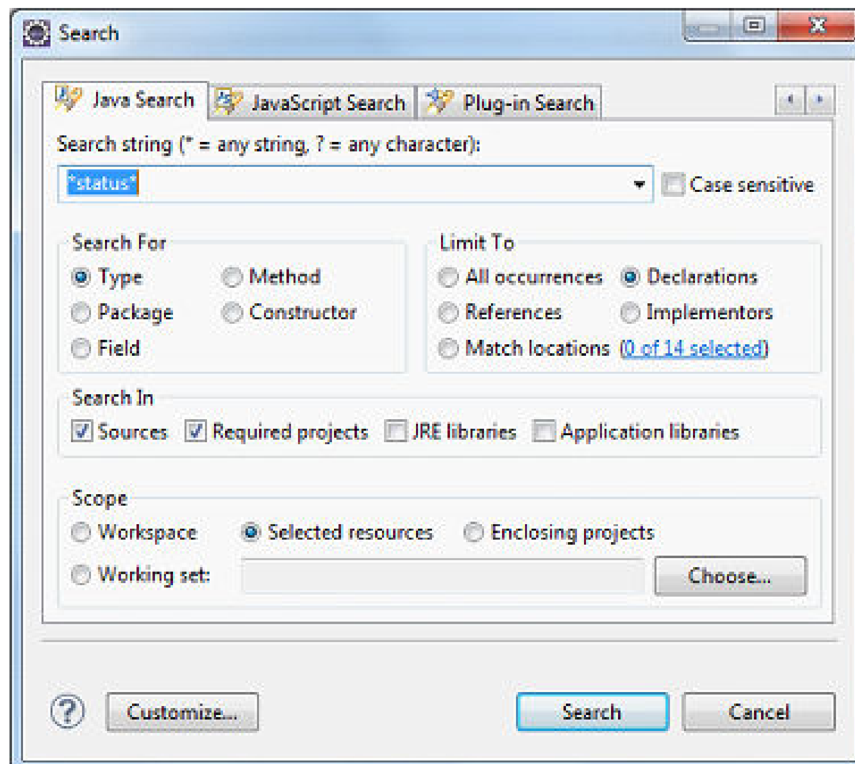
If you have any other comment, please write it below. Thank you.

Appendix F

Eclipse Tutorial

To answer the given comprehension questions, you need to parse the code and search for different types, methods and classes in the project. This is a brief tutorial explaining how to search, find a reference, open the hierarchy, reach the declaration of a function or type etc.

1. **Searching for an element corresponding to a specific pattern**:



To access this menu, **select the project**, then the Java search tab in the search menu of Eclipse, or simply press simultaneously **Ctrl+H**. This will bring up the search window, which has a series of tabs (you can choose any other tab according to what you are looking for). Enter a "**Search string**", select, in the "**Search For tab**", what you are looking for (a type, method, field, etc.). Then select in the "**Limit to**" tab what you are looking for (the declaration, references, etc.), select also in the "**Scope**" tab "**Selected resources**" and do not forget to select in the field "**Search In**": "**sources**" and "**required projects**" only!! to limit the search to the source files only.

Finally hit search. The search results will be displayed at the bottom of the screen. Double clicking on the results listed will focus the editor on that instance of the search string.

Notice: It is possible to use the wildcard * to replace any string.

Example: if you are looking for a class named like "Something -status- Something else", just write *status* in the "Search string" field, select "Type" in the "Search For tab", select "Declaration" in the "Limit To" tab and select in the "Scope" tab "Selected resources"; This will search all the classes that start by any string, contains essentially the word "status" and finish by any string.

Similarly, "?" replaces any character.

2. **Reach the declaration of a given element**:

   Select the desired element (method, field, class, etc.) then right click on it and choose "Open Declaration", or simply press **F3**.

3. **Open type hierarchy**:

   Displays the hierarchy of a type. Select the desired type or class, then right click on it and choose "Open Type Hierarchy", or simply press **F4**.

4. **Find references in the project**:

   Allows to search, in the whole project, all references related to an element. Just select the desired element (method, field, class, etc.) then right click on it and choose "References" then "Project".

5. **Outline of a class**:

   Displays the outline of a class by showing its different fields and methods. Select the desired class or object then right click on it and choose "Show In" then "Outline".