Université de Montréal

Patterns and Quality of Object-oriented Software Systems

par
Foutse Khomh

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

Avril, 2010

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Patterns and Quality of Object-oriented Software Systems

présentée par :

Foutse Khomh

a été évaluée par un jury composé des personnes suivantes :

Président-rapporteur	:	Julie Vachon
Directeur de recherche	:	Yann-Gaël Guéhéneuc
Membre du jury	:	Pierre-N. Robillard
Examinateur externe	:	Ahmed E. Hassan
Représentant du doyen de la FAS	:	Michel Delfour

Résumé

Lors de ces dix dernières années, le coût de la maintenance des systèmes orientés objets s'est accru jusqu' à compter pour plus de 70% du coût total des systèmes. Cette situation est due à plusieurs facteurs, parmi lesquels les plus importants sont: l'imprécision des spécifications des utilisateurs, l'environnement d'exécution changeant rapidement et la mauvaise qualité interne des systèmes. Parmi tous ces facteurs, le seul sur lequel nous ayons un réel contrôle est la qualité interne des systèmes. De nombreux modèles de qualité ont été proposés dans la littérature pour contribuer à contrôler la qualité. Cependant, la plupart de ces modèles utilisent des métriques de classes (nombre de méthodes d'une classe par exemple) ou des métriques de relations entre classes (couplage entre deux classes par exemple) pour mesurer les attributs internes des systèmes. Pourtant, la qualité des systèmes par objets ne dépend pas uniquement de la structure de leurs classes et que mesurent les métriques, mais aussi de la façon dont celles-ci sont organisées, c'est-à-dire de leur conception, qui se manifeste généralement à travers les patrons de conception et les anti-patrons.

Dans cette thèse nous proposons la méthode DEQUALITE, qui permet de construire systématiquement des modèles de qualité prenant en compte non seulement les attributs internes des systèmes (grâce aux métriques), mais aussi leur conception (grâce aux patrons de conception et anti-patrons). Cette méthode utilise une approche par apprentissage basée sur les réseaux bayésiens et s'appuie sur les résultats d'une série d'expériences portant sur l'évaluation de l'impact des patrons de conception et des anti-patrons sur la qualité des systèmes. Ces expériences réalisées sur 9 grands systèmes libres orientés objet nous permettent de formuler les conclusions suivantes:

- Contre l'intuition, les patrons de conception n'améliorent pas toujours la qualité des systèmes; les implantations très couplées de patrons de conception par exemple affectent la structure des classes et ont un impact négatif sur leur propension aux changements et aux fautes.

- Les classes participantes dans des anti-patrons sont beaucoup plus susceptibles de changer et d'être impliquées dans des corrections de fautes que les autres classes d'un système.
- Un pourcentage non négligeable de classes sont impliquées simultanément dans des patrons de conception et dans des anti-patrons. Les patrons de conception ont un effet positif en ce sens qu'ils atténuent les anti-patrons.

Nous appliquons et validons notre méthode sur trois systèmes libres orientés objet afin de démontrer l'apport de la conception des systèmes dans l'évaluation de la qualité.

Mots clés: patrons de conception, anti-patrons, modèles de qualité, apprentissage, réseaux bayesiens.

Abstract

Maintenance costs during the past decades have reached more than 70% of the overall costs of object-oriented systems, because of many factors, such as changing software environments, changing users' requirements, and the overall quality of systems. One factor on which we have a control is the quality of systems. Many object-oriented software quality models have been introduced in the literature to help assess and control quality. However, these models usually use metrics of classes (such as number of methods) or of relationships between classes (for example coupling) to measure internal attributes of systems. Yet, the quality of object-oriented systems does not depend on classes' metrics solely: it also depends on the organisation of classes, *i.e.*, the system design that concretely manifests itself through design styles, such as design patterns and antipatterns.

In this dissertation, we propose the method DEQUALITE to systematically build quality models that take into account the internal attributes of the systems (through metrics) but also their design (through design patterns and antipatterns). This method uses a machine learning approach based on Bayesian Belief Networks and builds on the results of a series of experiments aimed at evaluating the impact of design patterns and antipatterns on the quality of systems. These experiments, performed on 9 large object-oriented open source systems enable us to draw the following conclusions:

- Counter-intuitively, design patterns do not always improve the quality of systems; tangled implementations of design patterns for example significantly affect the structure of classes and negatively impact their change- and fault-proneness.
- Classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes than other classes.
- A non negligible percentage of classes participate in co-occurrences of antipatterns and design patterns in systems. On these classes, design patterns have a positive effect in mitigating antipatterns.

We apply and validate our method on three open-source object-oriented systems to demonstrate the contribution of the design of system in quality assessment.

Keywords: design patterns, antipatterns, quality models, machine learning, Bayesian Belief Networks.

Contents

List of Figures

List of Tables

List of Acronyms

DEQUALITE	*Design Enhanced QUALITy Evaluation*
DECOR	*DEtection&CORrection*
PADL	*Pattern and Abstract-level Description Language*
POM	*Primitives, Operators, Metrics*
PTIDEJ	*Pattern Trace Identification, Detection, Enhancement in* Java
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
XSLT	*Extensible Stylesheet Language Transformations*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
CART	*ClAssification and Regression Trees*
BBN	*Bayesian Belief Network*
IRS	*Issues Reporting System*
J2EE	*Java 2 Platform, Enterprise Edition*
EJB	*Enterprise JavaBeans*
CBO	*Coupling Between Objects*
NAS	*Number of Associations*

To my parents

To my brothers and sisters

Acknowledgements

Feeling gratitude and not expressing it is like wrapping a present and not giving it.
William Arthur Ward (1921–1994).

Many people accompanied me during the endeavor of my doctoral studies and of writing this dissertation. I am deeply grateful for their support. First of all, I would like to thank my supervisor, Yann-Gaël Guéhéneuc, for showing me the joy of research, for shaping my path to research by guiding me with his extensive knowledge, for teaching me how to write, for creating an enjoyable working environment, for his unending encouragement and support, and for becoming a friend.

My thankful admiration goes to Prof. Giuliano Antoniol, for all the essential advices and permanent guidance that I received from him. A great thanks goes also to Prof. Massimiliano Di Penta for all the technical knowledge he shared with me, for his professional advices, and for the fruitful discussions we have when we meet.

Special thanks go to Prof. Ahmed E. Hassan for enthusiastically accepting to be my external examiner and for reviewing my dissertation. I thank Prof. Pierre-N. Robillard for accepting to be on my committee and for reviewing my dissertation.

A Special thanks goes to Prof. Julie Vachon, who have been encouraging me all these research years, who was already on my pre-doctoral committee and who have enthusiastically accepted to chair my doctoral committee. Many thanks go to Stéphane Vaucher for all the interesting technical discussion, for his honest criticism, and suggestions, for the intensive but interesting time during paper writing, and for becoming a friend. A special thanks goes to my friend Wei Wu for his valuable comments on this dissertation.

I owe a lot of gratitude to Dieudonné Mayi for the enthusiasm with which he shared and enlarged my vision, for permanently motivating and helping me to go on. Thanks to my fellow doctoral students from the Ptidej and Soccer labs for their valuable comments and

suggestions. I thank my best friend, Carine, very much for supporting and encouraging me in whatever I do, for always being there when I need her, and for the wonderful time we always have. To my childhood friends Jean Bernard Chokouake and Hamida Yacoubou, for always believing in me. To my longtime friends Paul Samuel Njiki Njiki, Bruno Feunou Kamkui, Francis Didier Tatouchoup, Carine Tatouchoup, Anicet Kouomou Choupo, Jean François Djoufak Kengue, Romuald Momeya, Eric Tchouaket, Eric Bahel, Constant Lonkeng, Duplex Armand Tadem, Eddy Kwessi, Corine Michelle Kwessi and Hermann Fongang for all the fun we always have.

I am grateful to my parents, Dora and Abraham, for their love, for their unconditional support, and for believing in me–whenever and wherever. To my brothers and sisters.

—Foutse Khomh

Chapter 1

Introduction

1.1 Research Context: Software Quality

Large object-oriented software systems are now pervasive in our society. They are everywhere from document managers to embedded systems, such as navigation systems in aerospace, and they play a vital role in our life. Software systems are complex because they achieve many different, and often conflicting objectives and they comprise many components which are custom made and complex themselves [Bruegge and Dutoit, 1999]. Software systems are subject to constant change because users requirements are complex and evolve over time. However, this constant change is not without cost.

Systems, like people, get old [Parnas, 1994]. They increase in complexity and degrade in effectiveness [Lehman, 1996], unless the quality of the systems is controlled and continually improved; when the design of a system is poor, new changes to the system often degrade quality instead of improving it. Parnas attributed this degradation to a phenomenon that he called *Ignorant surgery*. This phenomenon occurs when changes are made on a system by people who do not understand its original design; these new changes sometimes invalidate the original design of the system and cause the design to degrade. Worst of all, after several of these changes, neither the original designers of the system, nor those who made the change understand the system. As a result, every new maintenance activity on the system becomes very expensive and the documentation increasingly inaccurate, making future changes even more difficult. Aging systems are often "faulty" because of error introduced when changes are made and the cost for removing these errors is high [Parnas, 1994]. Over the past two decades, maintenance have become the most time and resource consuming activity in the life cycle of systems. Maintenance costs have grown

to more than 70% of the overall costs of object-oriented systems [Pressman, 2001]; an increase due in part to aging systems.

Therefore, achieving good quality is essential to control and reduce the maintenance cost of object oriented systems. This goal require means to measure the quality of systems. However, quality has different meanings *eg.*, the capacity of a system to change at low cost, or the absence of bugs. In this dissertation, quality refers to change-proneness and fault-proneness. Quality attributes are non-functional requirements used to evaluate quality, *eg.*, maintainability and reliability.

Many quality models have been proposed to help assess the quality of object-oriented systems, *eg.*, [Dromey, 1996 ; McCall, 2001 ; Bianchi *et al.*, 2002 ; Bansiya and Davis, 2002 ; Zhu *et al.*, 2002 ; Singh and Goel, 2008]. A quality model is a set of quality attributes related to a set of metrics. The relationship between quality attributes and metrics specifies the quality evaluation process [ISO 9126, 1991]. The main goal of quality models is to facilitate the continuous improvement of a system [Boehm *et al.*, 1978 ; Dromey, 1996].

Briand and Wust [2002], who, surveyed quality models organised them in two categories: quality models based on mining software repositories and those based on experiments. They remarked that no quality model considered system design. They are all limited only to internal attribute of classes such as size, complexity and coupling or, at best, of pairs thereof, and disregard their organisation. Recently, some researchers have proposed process metrics from historical repositories [Moser *et al.*, 2008] and complexity metrics based on code change processes [Hassan, 2009], to improve quality assessment, but no study investigated the possible benefits of considering classes organisation, *i.e.*, system design, in the evaluation of their quality. Yet, the design of systems is the first thing developers should master when performing maintenance activities, if they want to avoid *Ignorant surgeries* that cause systems aging.

1.2 Problem Statement and Thesis

Design specifications are intensional and local [Eden and Kazman, 2003]. Their implementation results in located groups of classes with specific organisation in a system. The most popular forms of design implementations in object-oriented software systems are design patterns, code smells, and antipatterns. We use them as baseline for design in this dissertation.

Design patterns [Gamma *et al.*, 1994] are proven "good" solutions to recurrent design problems in object-oriented software design. They are used by developers either to generate an architecture [Beck and Johnson, 1994] or to enhance an architecture by superimposition [Hannemann and Kiczales, 2002]. Claimed advantages of design patterns include improved reusability, comprehensibility, and maintainability; the latter thanks to their supposed ability to make systems more robust to changes: "Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change" [Gamma *et al.*, 1994]. In practice, design patterns offer design motifs [Guéhéneuc and Antoniol, 2008]: ideal solutions that describe the roles played by classes to implement the motifs. Design motifs are an integral part of any reasonably well-developed system [Gamma *et al.*, 1994].

Antipatterns are conjectured in the literature to negatively affect the quality and evolution of systems [Brown *et al.*, 1998]. Antipatterns are "poor" solutions to recurring design problems, for example Brown's 40 antipatterns that describe common pitfalls in the software industry. Antipatterns are generally the result of a developer's lack of knowledge and–or experience in solving a design problem or applying some patterns: "something that looks like a good idea, but which backfires badly when applied" [Coplien and Harrison, 2005]. In practice, antipatterns relate to and manifest themselves as code smells, symptoms of implementation and–or design problems [Fowler, 1999] in source code. Antipatterns and design patterns concern the design of one or more classes.

Besides, very few studies quantitatively assesses the impact of design patterns and antipatterns on specific quality attributes. Existing studies limited to qualitative results, for example Venners [2005] claimed that design patterns improve the quality of systems while Wendorff [2001], suggested that their use do not always result in "good" designs; MacNatt and Bieman [2001] hinted that a tangled implementation of patterns may impacts quality negatively, and Du Bois *et al.* [2006] argued that refactoring of some antipatterns may improve understandability.

Vokac [2004] analysed the corrective maintenance of a large commercial system over three years and compared the fault rates of classes that participated in design motifs against those of classes that did not. Vokac's work is the best attempt to build a predictive model for fault-proneness based on design patterns. His work inspired this dissertation, in particular his use of logistic regression to analyse the correlations between design motifs and fault-proneness. Other studies deal with the changeability and resilience to change of design patterns [Aversano *et al.*, 2007] and of classes playing a specific role in design patterns [Di Penta *et al.*, 2008], or their impact on the maintainability of a large commercial

system [Wendorff, 2001] but neither did attempt to provide means to measure the quality of systems using the evaluation of the design of these systems. Nor did provide quantitative evidence of the claimed relation between design patterns, antipatterns, and quality.

These previous work lead us to the formulation of the following thesis:

Thesis:

Considering system design (design patterns and antipatterns) leads to more accurate quality models than using only the internal attributes of classes.

To verify our thesis, we propose and follow a method, DEQUALITE[1] (Design Enhanced QUALITy Evaluation), to build quality models that measure the quality of object-oriented systems by taking into account both their internal attributes, *i.e.*, the structure of their classes and their designs. To take into account the design of a system in an evaluation of quality, we should be able to characterise this design and understand its impact on quality. Therefore, through DEQUALITE, we perform a series of experiments intended for the evaluation of the impact of design patterns and antipatterns on the quality of systems.

1.3 DEQUALITE

DEQUALITE starts with the observation that the design of systems affects its quality *eg.*, when a design pattern implementation is recognised in a part of a system, the comprehension of classes involved in this implementation is improved. From this observation, DEQUALITE describes the following four steps, inspired by Dromey's approach [1995], to build a quality model mapping quality attributes of a system to its design.

1. Identify a set of quality attributes.
2. Identify and classify the most significant, tangible, quality-carrying design specifications of the system.
3. Propose a set of axioms for linking system design to quality attributes.
4. Evaluate the model, identify its weaknesses, and either refine it with new information on the system, like class metrics, or discard it and start again.

[1] DEQUALITE in French means good quality.

We use DEQUALITE as "fil conducteur" of the presentation of our contributions. Through the steps of DEQUALITE, we perform a series of experiments proving that design patterns and antipatterns impact the quality of systems. We build quality models that take into account design patterns and antipatterns and prove that they outperform state-of-the-art quality models built with class metrics only. The following section presents details on the steps.

1.4 Scenario: Building a Quality Model with DEQUALITE

At Step 1, we select the quality attributes change- and fault-proneness. Section 4.4 defines and explains their measurement. Remark that other quality attributes can also be consider with DEQUALITE.

At Step 2, we select design specifications: design patterns, and antipatterns. Once the design specifications are identified, we need to answer the following research questions:

1. *What is the impact of design patterns on the change- and fault-proneness of classes?* We investigate whether classes playing roles in design motifs are more change- or fault-prone than others.
2. *What is the impact of antipatterns on the change- and fault-proneness of classes?* We investigate whether classes participating in antipatterns are more change- or fault-prone than others.
3. *What is the interaction between antipatterns and design patterns and their impact on the change- and fault-proneness of classes?* We investigate the co-occurrence of antipatterns and design patterns on classes, and its effect *i.e.*, if this co-occurrence is for example related to higher or lower change-proneness *w.r.t.* antipatterns and design patterns alone.

We discuss these research questions in more details in the following chapters of this dissertation.

At Step 3, we use the results of Step 2 to propose and build Bayesian Belief Networks (BBNs) quality models taking into account design patterns and antipatterns.

At Step 4, we evaluate and refine the models from Step 3.

In summary, DEQUALITE describes steps to build and operationalise BBNs for the assessment of the quality of classes from their internal structure and their design.

1.5 Contributions

The main contributions of this dissertation are:

1. The development of a method, named DEQUALITE to systematically build quality models that take into account both the internal attributes of the systems and their design. This method permits us to build quality models that outperform state-of-the-art models built with class metrics only.

2. A study of the perceived impact of design patterns on quality. We found that design patterns are perceived by developers as not always improving the quality of systems.

3. An empirical study of the impact of playing roles in a (some) design pattern(s) for a class, on the internal (class metrics) and external (change- and fault-proneness) characteristics of classes. We found that playing roles in design motifs significantly affects the structure of classes as well as their change- and fault-proneness.

4. An empirical study of the impact of code smells on class change-proneness. We found that classes with code smells are significantly more likely to be subject to changes than other classes.

5. An empirical study of the impact of antipatterns on class change- and fault-proneness. We found that classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes than other classes.

6. An empirical study of the interaction between antipatterns and design patterns in systems. We found that when antipatterns and design patterns co-occur in a class, the negative effect of antipattern is mitigated.

1.6 Roadmap

The remainder of this dissertation provides the following content:

Chapter 2 (p. 8) reviews related work on quality models, design patterns, and antipatterns quality analysis

Chapter 3 (P. 22) reports our preliminary studies. We present and discuss the results of three pilots studies performed during our investigation of the relation between system design and quality.

Chapter 4 (P. 42) presents the setting of the experiments reported in this dissertation. This setting includes tools, variables and the statistical methods used in our experiments.

Chapter 5 (P. 56) presents a descriptive and analytic study of classes playing zero, one, or two roles in a (some) design pattern(s). The descriptive study shows that a non-negligible proportions of classes play one or two roles in design patterns. The analytic part of the study showing that internal (class metrics) and external (change- and fault-proneness) characteristics of classes are differently impacted by playing one and two roles.

Chapter 6 (P. 80) explores the impact of antipatterns on systems change- and fault-proneness, and provides quantitative evidence of the negative impact of antipatterns on classes change- and fault-proneness.

Chapter 7 (P. 104) investigates the interactions between design patterns and antipatterns in systems, and analyzed the effects of their co-occurrence in classes. Results suggest a positive effect of design patterns on antipatterns; design patterns mitigates the negative impact of antipatterns on classes.

Chapter 8 (P. 121) presents case studies of DEQUALITE and discuss the relevance of obtained quality models.

Chapter 9 (P. 136) presents the conclusions of this dissertation and outlines some directions of future research.

Appendix A (P. 160) presents the definitions of metrics used in this dissertation.

Appendix B (P. 163) presents the complete list of code smells and antipatterns considered in this dissertation with their definitions.

Appendix C (P. 167) presents detailed information on the distribution of antipatterns and design patterns in systems.

Chapter 2

Related Work

Shewhart, who was a statistician at AT&T Bell Laboratories in the 1920s', is regarded as the founder of statistical quality improvement and of the modern process improvement, which is based on his concept of process control, consisting in monitoring a process through the use of control charts [O'Regan, 2002]. In 1946, Deming, then working for the U.S. Occupation Force in Japan, applied these concepts to the manufacturing industry and the management of companies. Deming's work will lead to the Quality Movement that is best summarizes by Fujitsu's slogan "Quality built-in, with cost and performance as prime consideration".

In 1990, a growing interest in the quality assurance of software systems and the need of the software industry to standardize the evaluation of the quality of software systems led to the definition by the ISO (International Organization for Standardization) [1991] of a standard specifying six areas of importance for software evaluation (sometimes referred to as Quality Attributes, Software Metrics, or Functional and Non-Functional Requirements). These areas include: functionality, reliability, usability, efficiency, maintainability, and portability. For each area, the standard attempts to specify how they can be measured. The intention of this standard is to breakdown software systems into quality attributes that can be measured in terms of cost benefit, *i.e.*, quality attributes with a direct impact on systems' costs. However, the standard fails to provide means to measure these attributes. Therefore, many software metrics and software quality models have been introduced in the literature to help developers and quality analysts with the assessment of these quality attributes.

Briand and Wüst [2002] in their survey of quality models for object oriented systems found that although quality models have been introduced over the years, none of them

considered the organisation of classes in their evaluation of the quality of systems. They all focus on the evaluation of internal attributes of classes (such as size, filiation, and cohesion) or, at best, of pairs thereof. Moreover, except for the quality model introduced by Bansiya and Davis [2002], existing models do not link explicitly internal attributes to external quality characteristics. For example maintainability, which attempts to measure the effort required to diagnose, analyse, and apply a change to a system: most quality models propose measurements of internal attributes that influence the effort required to modify a system, like source code complexity, but fail to specify how maintainability is impacted by these internal attributes. This lack of explicit link renders difficult the practical use of these quality models because quality analysts and managers are first and fore most interested in external characteristics, like maintainability to plan development and maintenance activities.

Many principles and techniques exist to help developers design systems with good quality characteristics; among them, design patterns, which are "good" solution to recurring design problems, form an interesting bridge between internal attributes of systems, external quality characteristics, and software designs because they link internal attributes (concrete implementation of systems) and subjective quality characteristics (subjective perceptions on systems), such as reusability [Gamma *et al.*, 1994].

Since their popularisation in the software engineering community by the publication of the landmark book of Gamma *et al.* [1994], design patterns have gained importance for system design. The publication by Webster [1995] of the first book on "antipatterns", which are "poor" solutions to recurring design problems reinforced the importance of design styles.

Design patterns and antipatterns have been the subject of many studies aimed at their specification, detection, and analysis of their relation to software quality. Some authors like Venners [2005] claimed that design patterns improve the quality of systems, yet others, like Wendorff [2001], have suggested that their use do not always result in "good" designs. McNatt and Bieman [2001] hinted that a tangled implementation of patterns may impacts negatively quality while Du Bois *et al.* [2006] showed that the decomposition of the antipattern "god class" into a number of collaborating classes using well-known refactorings improves understandability.

In this chapter, we present the state-of-the-art on quality models, design patterns, and antipatterns' quality analyses. We start with an overview of quality models (Section 2.1). We also report work that studied design patterns and their relation to software quality (Sections 2.2) and work that studied antipatterns and their relation to software quality

(Section 6). We conclude this chapter with a discussion (Section 2.4) on the limitations of the presented work with regard to our thesis.

2.1 Quality Models

Zuse [1991] introduced the first approach to quantify the internal structure of procedural software systems. Since then, the large and rapid adoption of the object-oriented paradigm by the software engineering community and the multiple challenges introduced by it prompted researchers to turn their attention to the quality of object-oriented systems. In their review of quality models for object-oriented systems, Briand and Wüst [2002] organised this literature in two categories: mining software repositories and experiments. In addition to these two categories, we consider in the following also a third category of models of software quality attributes. We discuss some of the main work related to this dissertation along these three categories.

2.1.1 Mining Software Repositories

The Mining Software Repositories field analyzes data available in systems repositories and performs correlational studies to uncover interesting relations among variables defined on systems. These studies account for the majority of the studies on software quality. A situation that Briand and Wüst [2002] attributed to the fact that they are usually the only option in industrial settings. In software quality analysis, researchers generally use univariate and multivariate analysis to demonstrate the relationship between one or more measures of the structural properties of some systems and an external quality characteristic. In this case, the structural properties of the systems are the independent variables while the external quality characteristic is the dependent variable. There are three possible results to a correlational study: a positive correlation, a negative correlation, and no correlation. In the following, we focus on the more commonly used techniques in correlational studies: correlation coefficients (Pearson, Spearman, Kendall), negative binomial regression, logistic regression, linear ordinary least-squares regression, and exponential least-squares regression. Many of these techniques are often used both for univariate and multivariate analyses.

Correlations. Harrison *et al.* [1998] used Spearman's correlation to study the relationship between two coupling metrics: CBO [Chidamber and Kemerer, 1991] and NAS, which measures the Number of Associations between a class and its peers. NAS can be compute

directly from design documents, such as the Object Model of OMT [Harrison *et al.*, 1998]. They studied five object-oriented systems and found a strong relationship between CBO and NAS. They concluded that only one of these metrics is needed to assess the level of coupling in a system at design time. However, they argued that NAS could provide interesting early coupling estimates because it is available early in a system design, is simple to compute from design documents, and is easy to interpret. They also tested the relationship between coupling and understandability, the number of errors, and error density. For all the studied systems, they found no relationships between class understandability and coupling. They reported that limited evidence support a link between an increased coupling and an increased error density.

Negative Binomial. Weyuker *et al.* [2008] presented a study comparing the effectiveness of a negative binomial regression (NBR) model and a recursive partitioning (RP) model on three large industrial systems. Their intent was to find an alternative to negative binomial regression that could overcome the restrictions imposed by NBR both on linearity and additivity. Moreover, as well mentioned in their work [Weyuker *et al.*, 2008], there is ample evidence in the world that complex systems do not always obey simple models. To achieve their goal, they used the same predictor variables in two models (a NBR and a RP model) and found that the negative binomial model was able to identify files that contain 76 to 93 percent of faults while the recursive partitioning model identified files that contain 68 to 85 percent. They attributed their results to a possible insufficient tuning of the model fitting procedures but doubted that this shortfall of RP could be overcome by tweaking either the model, with for examples alternatives like C4.5, or by using specific parameters. They concluded their work by suggesting random forests as a more promising alternative. We plan to investigate that in future work.

Logistic Regression. Briand *et al.* [2002] presented a study assessing whether models of fault-proneness, based on design measurement, were applicable across systems, *i.e.*, if they could be viable decision making tools when applied from one object-oriented system to another. They selected two systems developed with a nearly identical development team, using a similar technology (object-oriented analysis and design and Java) but different design strategies and coding standards. To build the fault-proneness models, they first performed an univariate logistic regression analysis for each individual measure (independent variable) against the dependent variable, *i.e.*, no fault/fault. Second, they build multivariate prediction models using logistic regression and Multivariate Adaptive Regression Splines (MARS). They justified the combination of MARS and logistic regression by

the need to achieve high accuracy and obtain models with more realistic functional forms. The validation of their models suggested that a model generated by MARS outperforms logistic regression models where the relationship between the logit and the independent variables is linear (log-linear model). They also introduced a cost-benefit model and applied it to assess the economic viability of fault-proneness models as a function of factors, such as defect detection effectiveness. They applied a fault-proneness model built on one of the systems to the other system and concluded that it can provide substantial benefits, even when used across systems.

Linear Ordinary Least-Squares Regression. Brito e Abreu and Melo [1996] presented a study evaluating the impact of object-oriented design on software quality attributes (defect density and rework). They used the MOOD metric suite to measure the use of object-oriented design mechanisms. They collected data from the development of eight small-size C++ information management systems based on identical requirements and assessed the referred impact (*i.e.*, impact of object-oriented design on defect density and rework) by means of the linear ordinary least-squares regression and the Pearson correlation. They used the coefficient of determination (R^2) to evaluate their model. They discussed how object-oriented design mechanisms, like inheritance, polymorphism, information hiding, and coupling, influenced the defect density and rework of systems. They also built some predictive models based on object-oriented design metrics and concluded on the importance of quantifying the influence of object-oriented design mechanism on quality, because it can help in the training of novice designers by the means of heuristics [Brito e Abreu *et al.*, 1995] embedded in design tools and also project managers during the planning process.

Exponential Least-Squares Regression. Misic and Tesic [1998] presented an empirical study looking for appropriate measures of quality and establishing simple, usable, and cost-effective models for the control and estimation of the quality of object-oriented systems. They performed their study on a set of seven systems developed in a small software company within a period of 18 months. They collected data on the internal characteristics of the systems and performed an analysis in two steps: first, they applied a Pearson correlation analysis to identify candidate measures of structural properties with strong mutual dependence. Second, they used simple and exponential least-squares regression to derive a number of models for pairs of measures that were found to be correlated. They computed the R^2 of each model and selected the models suitable for estimation. They discussed these models and suggested that external characteristics such as “effort” corre-

lates well with the total number of classes and the total number of methods. However, they concluded on the necessity to apply their results with care as they were obtained on small-size systems.

2.1.2 Experiments

Experiments in the context of software quality are studies that attempt to demonstrate relations between some structural properties of systems (independent variables) and some external characteristics, like understandability (dependent variable). In these studies, subjects are required to undertake some software development tasks on a set of systems during which the structural properties of the systems are controlled and the performance of the subjects measured. Briand and Wüst [2002] in their survey of quality models found that controlled experiments were far fewer in number than correlational studies.

The first experiment on the quality of object oriented systems was presented by Lake and Cook [1992]. Their study attempted to relate flat and deep inheritance structure to understandability, modifiability, and debugability. Their experiment was performed with two groups of five and six students that were asked to perform three different tasks on four C++ systems (an APL compiler, Borland's C++ library, some C++ instructional code, and an accounting system). The measurements of these systems revealed that they generally consisted of few unconnected classes (less than 20) with little use of inheritance. Only in few cases, the systems were having inheritance trees with depth three or more, the majority of classes being at the same level. The result of their experiment suggested that developers more effectively debug and modify classes with lower depth of inheritance. Since then, other controlled experiments have investigated relations between some aspects of system understandability and maintainability and inheritance [Daly *et al.*, 1996 ; Harrison *et al.*, 2000].

Daly *et al.* [1996] performed a series of subject-based laboratory experiments, including an internal replication, to test the effect of depth of inheritance on the maintainability of object-oriented systems. The study was performed on two object-oriented systems implemented in C++. Both were simple database systems. Two versions of each system were used, a version with inheritance and a version with no inheritance (flat). 31 students enrolled in an object-oriented programming course participated in the experiments. First, the subjects were timed performing identical maintenance tasks on a system with a hierarchy with three levels of inheritance and on the equivalent system with no inheritance. Second, a replication was performed. Third, a second experiment with a design similar to the first one was performed. Again, subjects were timed performing identical maintenance

tasks on a system with a hierarchy with five levels of inheritance and on the equivalent system with no hierarchy. The results of these experiments showed that a system with three levels of inheritance is easier to modify (*i.e.*, takes less time) than a system with no inheritance. However, a system with five levels of inheritance takes longer to modify than a system without inheritance. The depth of inheritance thus has an impact on systems maintenance.

Harrison *et al.* [2000] replicated Daly's study. They analysed a C++ system without any inheritance and a corresponding system containing three levels of inheritance, as well as a second larger C++ system without inheritance and a corresponding system with five levels of inheritance. Both systems were modeling databases for a University human resource system. Their results suggested that systems without inheritance were easier (*i.e.*, took less time) to modify than systems with either three or five levels of inheritance. However, they also pointed out that while modifying a flat system is easier than modifying one with five inheritance levels, one should be careful to imply that in general, the systems without inheritance are easier to understand. They claimed that the size and functionality of a system has a greater impact on its understandability than the amount of inheritance used.

Wood *et al.* [1999] studied the structure of object-oriented C++ systems to assess the relation between the use of inheritance and software maintenance. They concluded that the use of inheritance in object-oriented systems may inhibit software maintenance. In their study, the authors used a multi-method approach based on structured interviews, surveys, and controlled experiments. They justified the use of this approach by arguing that the combination of complementary techniques led to more robust conclusions and greater understanding of the factors lying behind empirical results.

2.1.3 Models of Software Quality Attributes

This section presents work that proposed models to assess object-oriented quality attributes, such as reusability, quantitatively. In these studies, the overall quality of systems is defined through a set of quality attributes. Relations between these attributes are defined and metrics are proposed to measure them. However, there is not yet a consensus on what are important quality attributes. The relations among different quality attributes is still unclear. Many quality models have been introduced in the literature, each attempting to define quality attributes so that they can be measured objectively. The vast majority of these quality models are hierarchical models to avoid overlaps between quality attributes. In the following, we discuss the main quality models defined in the literature.

McCall's Quality Model. The first quality model was introduced by McCall *et al.* [1977] at the US Air-Force electronic system decision department. The rationale of this model was to measure the relation between internal and external quality characteristics of a system. The quality attributes defined in the model were selected to reflect both the users' views and the developers' priorities. Therefore, three major perspectives were considered in their definition and selection: software revision, software transition, and software operations. Software revision includes the ability of systems to undergo changes, their maintainability, flexibility, and testability. Software transition concerns the portability, the reusability, and the interoperability while software operations correctness, reliability, efficiency, integrity, and usability. McCall's model organised these three external quality characteristics in a hierarchy of factors, criteria, and metrics, which were decomposed in 11 attributes describing the external view of the system, as viewed by the users. The model also introduced 23 quality attributes to describe the internal characteristics of the system, as perceived by developers. A set of metrics was defined and used to provide a scale and method for measurement. The metrics were computed by people answering "yes" or "no" questions. The main contribution of this model was to provide a baseline relation between quality characteristics and metrics. However, because of its subjective measurement of quality, relaying on people answering questions, McCall's model has been largely criticized and not used in practice.

Boehm's Quality Model. Following McCall's quality model, Boehm [1976 ; 1978] proposed a model that attempted to qualitatively define the quality of a software through a given set of attributes and metrics. Boehm's model is similar to McCall's in that it also organised the quality attributes in a hierarchy with high-level characteristics, intermediate-level characteristics, and primitive characteristics. It added some more characteristics to McCall's model, emphasizing maintainability and hardware performance. Boehm's model targeted various dimensions, considering the types of users expected to be working with the system [Vinayagasundaram and Srivatsa, 2007]. General utility (high-level characteristic) was refined into portability, utility, and maintainability, which represented basic high-level requirements of actual use. The utility was further refined into reliability, efficiency, and human engineering, while maintainability was refined into testability, understandability, and modifiability. The intermediate-level characteristic included seven quality attributes that Boehm claimed to represent the attributes expected from a system: portability, reliability, efficiency, usability, testability, understandability, and flexibility. The primitive characteristics provided the foundation for defining quality metrics. While McCall's model primarily focused on the precise measurement of high-level characteristics, Boehm's quality

model explored a wider range of characteristics with an extended and detailed focus on maintainability. Boehm's model is considered an important predecessor of todays' quality models

ISO/IEC 9126-1. In 1991, in an attempt to standardize the evaluation of software systems, the International Organization for Standardization (ISO) proposed the ISO 9126 standard that is divided into four parts addressing respectively: quality model; external metrics; internal metrics; and quality-in-use metrics [ISO 9126, 1991]. ISO 9126 Part One, referred to as ISO/IEC 9126-1, specifies six areas (quality characteristics) of importance for software quality evaluation and provides, for each of these areas, specifications that attempt to make them measurable (these characteristics are broken into sub-characteristics). The six areas are: functionality, reliability, usability, efficiency, maintainability, and portability. The main limitation of this model is that it does not show clearly how these sub-characteristics can be measured. However, it has the advantage of clearly identifying internal attributes and external characteristics of systems.

Dromey's Quality Model. In an attempt to build a model broad enough to work for different kinds of systems, Dromey proposed a quality model recognizing that "a more dynamic idea for modeling the process is needed to be wide enough to apply for different systems" [Dromey, 1995 ; Dromey, 1996] because quality evaluation differs for each system. In this model, Dromey focused on the relationship among quality attributes and their sub-attributes and attempted to connect system properties with quality attributes. The principal elements of Dromey's model were:

- Product properties that influence quality.
- High-level quality attributes.
- Means of linking the system properties with the quality attributes.

Dromey's quality model is built on a five-step process:

1. Choose a set of high-level quality attributes necessary for the evaluation.

2. List all the components/modules in the system.

3. Identify quality-carrying properties for the components/modules (properties of the component that have the most impact on the software quality attributes from the list created in the previous steps).

4. Determine how each property affects the quality attributes.

5. Evaluate the model and identify its weaknesses.

Dromey's Quality Model introduced eight high-level quality characteristics, the six from ISO/IEC 9126-1 and reusability and process maturity [Vinayagasundaram and Srivatsa, 2007]. The most important issue with this model is his primary focus on the software system, specifically the code.

QMOOD Quality Model. In 2002, based on Dromey's work [Dromey, 1995 ; Dromey, 1996], Bansiya and Davis [2002] introduced QMOOD, a hierarchical model for the assessment of the quality of object-oriented systems. Their quality model consisted in six equations that established relationships between six object-oriented design quality characteristics (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) and 11 identified structural design properties of the object-oriented paradigm among which: encapsulation, coupling, polymorphism, data abstraction, and hierarchies. A set of object-oriented metrics was introduced to measure these design properties. This model has been validated on large industrial systems: Microsoft Foundation classes (5 versions), Borland Object Windows Library (4 versions), and 14 versions of a medium-size industrial system written in C++ and implementing an interpreter for a fictitious language named "COOL". A key characteristic of the QMOOD model was that "it could be easily modified to include different relationships and weights". The authors claimed that it provided "a practical quality assessment tool adaptable to a variety of demands". QMOOD is the most used model and also the most referenced model in recent studies.

2.2 Studies on Design Patterns and Quality

Since their introduction by Gamma *et al.* [1994], there has been a growing interest in the use of design patterns. Many pieces of work are related to design patterns, from their definition [Kampffmeyer and Zschaler, 2007] to their identification [Guéhéneuc and Antoniol, 2008]. We present here work on the impact of design motifs (*i.e.*, concrete solutions of design patterns) on object-oriented quality.

Bieman *et al.* [2001a ; 2003] examined common recommended programming styles, including design patterns, on several different software systems and concluded that in contrast with common lore, the use of design patterns could lead to more change-prone classes rather than less change-prone classes during evolution. With McNatt, Bieman also

performed a qualitative study of the coupling between motifs [2001] and claimed that, when motifs were loosely composed and abstracted, maintainability, modularity, and reusability were well supported by the motifs. However, they concluded on the need for further studies to examine different motif compositions and their impact on quality.

Di Penta *et al.* [2008] studied the change-proneness of classes playing different roles and the kinds of changes affecting these classes. Their results confirmed the expected, theoretical impact of motifs. For example, they found that in Abstract Factory, classes playing concrete roles changed more often than those playing abstract roles. They also highlighted deviations from the intuition, in the case of Composite for example, in which classes playing the role of Composite can be complex and undergo many changes. In their other study [Aversano *et al.*, 2007], they focused on resilience to change and concluded that design motifs changed frequently and that the amount of co-change did not depend on the motif, but on the roles played by the motif to support the systems features. They also reported that design motifs were often changed either in their implementation or by adding subclasses or changing method interfaces; which caused a higher co-change on client classes.

Hannemann and Kiczales [2002] studied the use of aspect-oriented programming and show that 17 of the 23 design patterns [Gamma *et al.*, 1994] could benefit from their "aspectisation" to overcome: the impact of motifs on systems and of systems on motifs; the loss of motif modularity and of traceability; the invasiveness of motifs; the difficulty to reason about classes involved in several motifs. They proposed AspectJ implementations of the motifs that they claimed to "better align dependencies in the code with dependencies in the solution structure".

Lange and Nakamura [1995] demonstrated that design patterns could serve as a guide in system exploration and thus make the process of system understanding more efficient. Through motif-driven system exploration, they showed that if design motifs were recognized at a certain point in the understanding process, they helped in "filling in the blanks" and in further exploring a system, thus improving the understandability of the system. However, this study was limited to the Observer, Composite, and Decorator patterns.

Masuda *et al.* [1999] studied the impact of applying design patterns in systems. They implemented a set of systems with two releases for each: one release using design patterns, the other not using design patterns. Using the C&K metrics, they showed that there is no statistically significant impact of applying patterns and suggested that new, more appropriate metrics should be devised for pattern-based systems.

Ng *et al.* [2007] investigated whether design motifs deployed in software systems were used or not by maintainers when doing the following tasks: T1 adding a new class as a concrete participant, T2 modifying the existing interfaces of a participant, and T3 introducing a new client. Their study focused on perfective maintenance as they claimed that it is the most common maintenance activity. They divided design motifs along three types of programming elements: concrete participants, abstract participants, and clients. They claimed that performing an anticipated change typically entails the completion of one or more of the tasks T1, T2, and T3, respectively. Their experiment involved 215 subjects who were asked to perform 6 anticipated changes in 3 systems with a total of 17.8 KLOC and over 230 classes in 12 packages. The experiments involved six design patterns that together cover creational, structural, and behavioral patterns. The results of these experiments showed that almost all subjects performed T1, a majority of the subjects performed T3, but, on average, only about half of the subjects performed T2. These results suggested that the tasks differed in popularity when a maintainer completes an anticipated change. The authors further found that the code implemented by the subjects who used the deployed design motifs (by performing tasks T1, T2 and T3, respectively) were significantly less faulty than the code done by subjects who did not used them. Suggesting a positive impact of design patterns on fault-proneness.

Tatsubori *et al.* [1998] proposed one of the first attempt to integrate design motifs directly in a programming language. They used the OpenJava meta-object protocol to extend the Java programming language with a new syntax to express straight-forwardly design motifs in source code without the complexity involved in giving all implementation details. They illustrated their approach with the Adapter pattern and the Visitor pattern written in OpenJava. They claimed that systems with comments are more understandable than systems using design patterns and concluded that there is little evidence on a positive impact of design patterns on quality.

Vokac [2004] analysed the corrective maintenance of a large commercial system. He performed an automatic identification of design motifs on weekly snapshots of this system over a period of three years and compared fault rates for classes playing roles in design motifs with other classes. Classes in motifs were less fault-prone than others with differences in fault rates ranging from 63 percent to 154 percent on average. He also noticed that the Observer and Singleton motifs are correlated with larger classes; classes playing roles in Factory Method were more compact, less coupled, and less fault-prone than others classes; and, no clear tendency existed for Template Method. His work provided the first quantitative evidence of a relationship between design motifs and class fault-proneness.

Wendorff [2001] evaluated the use of design motifs in a large commercial software systems. They discussed two categories of inappropriate implementations of design patterns. In the first category, patterns were simply misused by developers who had not understood their rationale. The second category contained patterns that do not fall into the first category, but which do not match the project's requirements. They analyzed this second category and found the inappropriate implementations to be caused by: developers who overestimated the future volatility of requirements and opted for motifs to build flexibility; changes in requirements over the lifetime of the system that caused motifs to become obsolete; application of patterns without any regard for the quality goals of the system (for example, a software developer wanted just to gain some experience with some patterns); and the addition of useless features caused by a desire to embellish implementation of design patterns and make them look like those in the book. They concluded that design patterns do not improve a system design necessarily, that a design could be over-engineered [Kerievsky, 2004], and that the cost of removing patterns is high.

Wydaeghe *et al.* [1998] presented a study on the concrete use of six design patterns when building an OMT editor. They discussed the impact of these patterns on reusability, modularity, flexibility, and understandability. They also discussed the difficulty of the concrete implementation of these patterns. They concluded that although design patterns offer several advantages, not all patterns had a positive impact on quality attributes. However, this study was limited to the authors' own experience and thus their evaluation of the impact of these patterns on quality could hardly be generalized to other contexts of development.

2.3 Studies on Antipatterns and Quality

Brown [1998] described antipatterns as the result of developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. He suggested that antipatterns make maintenance harder, decreasing the quality of systems. He did not provide any quantitative evidence to support his claim.

The first work to investigate quantitatively the relation between antipatterns and quality were [Ignatios *et al.*, 2003 ; Ignatios *et al.*, 2004], who performed controlled experiments with 20 students on two systems to understand the impact of God Classes on the understandability and maintainability of systems. The results of their study suggested that God Classes affect the evolution of design structures and considerably affects the subjects' use of inheritance. Du Bois *et al.* [2006], described later that the decomposition of these God

Classes into a number of collaborating classes, using well-known refactorings, improved the maintainers' comprehension of these classes.

Wei and Raed [2007] investigated the relationship between the probability of a class to be faulty and some antipatterns based on three versions of Eclipse and showed that classes with the antipatterns God Class, Shotgun Surgery, and Long Method have a higher probability to be faulty than other classes. They concluded on the need for broader studies to validate their results.

Recently, Olbrich *et al.* [2009], analysed the historical data of Lucene and Xerces over several years and concluded that God Classes and Shotgun Surgery have a higher change frequency than other classes; with God Classes featuring more changes. They neither performed an analysis to control the effect of the size on their results nor studied the kinds of changes affecting these antipatterns. We address this limitation in Chapter 6.

2.4 Summary

This previous work raised the awareness of the software-engineering community about the impact of design patterns and antipatterns on software quality. In this dissertation, we build on this previous work and propose more detailed and extensive empirical studies of the impact of design patterns, code smells, and antipatterns on code evolution phenomena. Our aim is to analyse the impact of design patterns and antipatterns on quality and build quality models that assess the quality of object-oriented systems by taking into account both the internal attributes of the systems and these design styles. Although some work have assessed some architectural characteristics of systems [Brito e Abreu and Melo, 1996 ; Daly *et al.*, 1996 ; Harrison *et al.*, 1998 ; Wood *et al.*, 1999], none has provided predictive quality models taking into account the design of systems.

To date, no method exists to help build quality models taking into account the design of systems. No quality models presented in the literature has assessed the design quality of systems directly [Briand and Wüst, 2002]. These quality models all used class-based metrics or metrics on pairs of classes. In this dissertation, we propose a method called DEQUALITE, to build quality models that measure the quality of object-oriented systems by taking into account both their internal attributes and their design.

Chapter 3

Pilot Studies

This chapter presents the results of three pilots studies performed during our investigation of the relations between systems design and quality. These results are important to better understand the choices made in this dissertation.

Eden *et al.* [2003] with their "Intension/Locality thesis" defined design specifications as "local and intensional", *i.e.*, specifications which are abstract in the sense that they can be formally characterised by the use of logic variable but are limited only to a part of the system. Design concretely manifests itself in systems through design styles such as: design patterns, code smells, and antipatterns. Code smells are generally considered more low-level and symptomatic antipatterns. In fact, the presence of some specific code smells can, in turn, manifest in antipatterns [Brown *et al.*, 1998], of which code smells are parts of.

Therefore, to achieve our goal of taking into account systems design in quality evaluations, we use design patterns, code smells, and antipatterns as baseline for design and investigate the existence of relations between them and system quality. The pursuit of this investigation led us to perform three pilot studies.

The first pilot study concerns the relation between design patterns and quality. We surveyed software developers to assess their perceived impact of the 23 design patterns from the catalog by Gamma *et al.* [1994] on the quality of systems. The result of this study revealed a relation between design patterns and quality and also the need for more extensive analysis of this relation. In Chapter 5, we perform experiments to further analyse and quantify this relation.

In the second pilot study, we investigated the relation between code smells and classes change proneness. The general perception on code smells is that they hinder the main-

tenance and evolution of systems [Fowler, 1999]. Finding a relation between these code smells and system quality would bring evidence to the possible existence of a relation between antipatterns and quality. In this pilot study, instead of performing a survey, we performed an experiment to assess their relation with classes change-proneness. The results of this study revealed a strong relation between code smells and change proneness and laid the ground for the broader study on the relation between antipatterns and quality presented in Chapter 6.

The third pilot study concerned the fault proneness of classes generally measured using data obtained by merging information extracted from issues reporting systems, such as Bugzilla, and versioning systems, such as Concurrent Version System (CVS). Most quality modeling studies presented in the literature currently use this approach. For example, Gyimóthy *et al.* [2005] report an empirical study in which CVS and Bugzilla data are used to identify error-prone classes. However, many work have recently casted doubts about the quality of data contained in these repositories [Ayari *et al.*, 2007 ; Bird *et al.*, 2009]. Some have pointed out the negative impact bias data may have on predictive activities, *eg.*, [Bird *et al.*, 2009] found that although some approaches for building models in the presence of bias do exist, bias in data significantly affects the performance of prediction models. These concerns about the quality of data stored in issues reporting systems left us with the following question: Do Issues Reporting Systems report faults? In our third pilot study, we answered this question through the analysis of three issues reporting systems.

The remaining of this chapter summarizes the key results of our three pilot studies and highlights our choices for next studies. We only present key results for the sake of completeness and simplicity. Interested readers can find details in our referenced papers [Khomh and Guéhéneuc, 2008a ; Antoniol *et al.*, 2008 ; Khomh *et al.*, 2009a].

3.1 Pilot Study 1: Do Design Patterns Impact Quality?

The intuition behind this pilot study is that design patterns have an impact on the quality of systems. Our hypothesis follows common lore: *H_0: design patterns impact software quality positively.* Our goal is to quantify and qualify this impact to confirm or refute the hypothesis.

Development and maintenance are manual activities performed by engineers. Thus, engineers' evaluation is important. In this study, we chose to survey developers to assess their perceived impact of design patterns on the quality of their systems.

3.1.1 Definition of the Questionnaire

Following previous work [Gamma *et al.*, 1994 ; Bansiya and Davis, 2002 ; Guéhéneuc *et al.*, 2005], we chose the following set of quality attributes, based on their relevance to design patterns:

- Attributes related to design:
  - **Expandability:** The degree to which the design of a system can be extended.
  - **Simplicity:** The degree to which the design of a system can be understood easily.
  - **Reusability:** The degree to which a piece of design can be reused in another design.
- Attributes related to implementation:
  - **Learnability:** The degree to which the code source of a system is easy to learn.
  - **Understandability:** The degree to which the code source can be understood easily.
  - **Modularity:** The degree to which the implementation of the functions of a system are independent from one another.
- Attributes related to runtime:
  - **Generality:** The degree to which a system provides a wide range of functions at runtime.
  - **Modularity at runtime:** The degree to which the functions of a system are independent from one another at runtime.
  - **Scalability:** The degree to which the system can cope with large amount of data and computation at runtime.
  - **Robustness:** The degree to which a system continues to function properly under abnormal conditions or circumstances.

Each quality attribute was evaluated by participants to the survey using a six-point Likert scale:

A - Very positive.

B - Positive.

C - Not significant.

D - Negative.

E - Very Negative.

F - Not applicable.

The sixth value allowed developers not to answer a question if they did not know or were not sure about the impact of a design pattern on a quality attribute.

For every design pattern in [Gamma *et al.*, 1994] and for every quality attribute from our set, the developers were asked to assess the impact of the pattern on the quality of a system in which the pattern would be used appropriately, as they would during a technical review [Pressman, 2001] or possibly while performing a program comprehension-related activity during maintenance and evolution. The questionnaire is available in [Khomh and Guéhéneuc, 2008b] and online[1].

3.1.2 Data Collection and Processing

We collected developers' evaluations between January and April 2007 by posting our questionnaire on three mailing lists, *refactoring*, *patterns-discussion*, and *gang-of-4-patterns*.

Among the many answers that we received, we selected the questionnaires of 20 developers with a verifiable experience in the use of design patterns in software development and maintenance. This number of collected evaluations is larger than in any previous work.

A pre-analysis led us to realise that the differences between `Positive` and `Very Positive` answers were due to some developers being less strict than others and, thus, their `Very Positive` evaluations were not directly relevant. This fact has been confirmed in discussions with the developers. For example, for Builder and expandability, we had 19% of developers considering the pattern `Very Positive` while 63% considered it `Positive` and 18% considered it `Neutral`. Therefore, we chose to aggregate answers `A` and `B` and answers `D` and `E`: `Positive = A and B`, `Neutral = C`, and `Negative = D and E`.

Using the previous three-point Likert scale, we computed the frequencies of the answers on each quality attribute: `Positive`, `Neutral`, and `Negative` and we carried out a Null hypothesis test to assess the perceived impact of the patterns on the quality attributes. Answers `F` were not considered because they represented situations where the developers did not know or did not want to evaluate the impact of a pattern.

[1] `http://khomh.net/experiments/thesis/`

Table 3.1 – Estimation of the impact of the three design patterns on quality attributes.

Attributes	Composite		A.Factory		Flyweight	
	E	R(%)	E	R(%)	E	R(%)
Expendability	+	0.00	+	0.00	−	1.76
Simplicity	+	5.92	+	30.36	−	0.00
Reusability	+	15.09	+	50.00	−	15.09
Learnability	+	1.76	−	15.09	−	0.00
Understandability	+	5.92	−	15.09	−	0.00
Modularity	+	5.92	+	0.37	−	5.92
Generality	+	1.76	+	1.76	−	0.15
Mod. at Runtime	+	30.36	−	30.36	−	0.15
Scalability	−	30.36	−	1.76	+	1.76
Robustness	−	0.15	−	0.00	−	1.76
	8 + / 2 −		5 + / 5 −		1 + / 9 −	

3.1.3 Analyses and Results

We now summarize the results obtained for three design patterns: Abstract Factory, Composite, and Flyweight, and the three quality attributes mentioned by the GoF [Gamma *et al.*, 1994, page xiii]: reusability, expandability, and understandability.

We chose to present results for this three design patterns first because of their popularity—they are among the most commonly used patterns—and second because they appear to be globally considered as positive, neutral, and negative.

3.1.3.1 Quantitative Results

Using the results collected from the questionnaires, we carried out Null hypothesis tests to quantify the impact of the design patterns on the quality attributes and then confirm or refute the hypothesis H_0.

The Null hypothesis test yields the results summarized in Tables 3.1, and 3.2. Full results for all patterns from [Gamma *et al.*, 1994] and details of the test can be found in [Khomh and Guéhéneuc, 2008b]. In these tables, the sign + means that, with our Null hypothesis test, the impact of the pattern on the quality attribute is positive else the sign is − (it can be negative or neutral). The number next to a sign represents the risk of making this decision. We computed this risk using the cumulative density of the Bernoulli distribution.

We now present a brief qualitative discussion of the results.

Table 3.2 – Estimation of the impact of design patterns on the three quality attributes.

Design Patterns	Expendability(%)		Understandability(%)		Reusability(%)	
	E	R(%)	E	R(%)	E	R(%)
A.Factory	+	0.00	−	15.09	+	50.00
Builder	+	0.15	+	0.37	−	15.09
F.Method	+	1.76	−	30.36	+	15.09
Prototype	+	30.36	+	30.36	+	30.36
Singleton	−	0.15	+	0.15	−	0.37
Adapter	+	30.36	−	30.36	+	5.92
Bridge	+	0.37	+	50.00	−	30.36
Composite	+	0.00	+	5.92	+	15.09
Decorator	+	0.15	−	30.36	−	5.92
Facade	+	30.36	+	1.76	−	5.92
Flyweight	−	1.76	−	0.00	−	15.09
Proxy	−	30.36	−	5.92	+	50.00
Ch.Of.Resp	+	0.15	−	5.92	+	30.36
Command	+	5.92	−	5.92	−	5.92
Interpreter	+	5.92	+	5.92	+	30.36
Iterator	+	0.15	+	50.00	+	5.92
Mediator	+	30.36	+	30.36	−	1.76
Memento	−	5.92	−	30.36	−	15.09
Observer	+	0.15	−	30.36	+	50.00
State	+	5.92	+	30.36	−	1.76
Strategy	+	1.76	+	15.09	−	30.36
T.Method	+	0.37	−	15.09	+	30.36
Visitor	+	5.92	−	1.76	−	1.76
	19 + / 4 −		11 + / 12 −		11 + / 12 −	

Composite. By analysing Table 3.1, it appears that the Composite pattern is mostly perceived as having a positive impact on the quality of systems. All quality attributes are impacted positively but for scalability and robustness, which are considered neutral. Given the purpose of the Composite pattern, having a neutral impact on scalability is rather surprising.

Abstract Factory. Table 3.1 shows that half the quality attributes are considered as positively impacted while the other half is not. It is not surprising that the pattern is overall judged as neutral given its purpose and complexity. What is surprising, however, is that learnability and understandability are felt negatively impacted.

Flyweight. Table 3.1 reports that this pattern is perceived as impacting negatively all quality attributes but scalability. Given the purpose of the pattern, it is not surprising that

its impact on scalability is judged positively. The negative perception could be explained by the less frequent use of Flyweight in comparison with Composite and Abstract Factory.

Expandability. Table 3.2 presents the analysis of the developers' evaluations of the impact of the design patterns on expendability. All developers felt that expandability is improved when using patterns, in conformance with previous claims [Gamma *et al.*, 1994].

Reusability. Table 3.2 shows that reusability is felt as being slightly more negatively impacted by design patterns, with 13 neutral or negative patterns and 10 positive patterns. This result is rather surprising as the use of patterns is claimed to improve reusability [Gamma *et al.*, 1994].

Understandability. Table 3.2 presents the analysis of understandability. Similarly to reusability, developers felt that understandability was rather slightly negatively impacted by the use of patterns.

3.1.4 Summary

This pilot study revealed that design patterns are perceived by developers as having an impact and that they do not always improve the quality of systems. Some patterns are reported to decrease some quality attributes and to not necessarily promote reusability, expandability, and understandability. This result confirms the intuition behind our work that design patterns do impact the quality of systems. This pilot study also revealed the need for more detailed studies to assess this impact of design patterns on the quality of systems. In Chapter 5, we analyse and quantify the impact of some single and tangled design patterns implementations on the internal characteristics of classes and evolution phenomena (change- and fault-proneness).

3.2 Pilot Study 2: Do Code Smells Impact Change-proneness?

Code smells are "poor" solutions to recurring implementation problems, opposite to idioms [Coplien, 1991] and, to some extent, to design patterns [Gamma *et al.*, 1994], because they pertain to implementation while design patterns pertain to design. In practice, code smells are in-between design and implementation: they may concern the design of a class, but they concretely manifest themselves in the source code as classes with specific imple-

mentation. They are usually revealed through particular metric values [Marinescu, 2004]. At a higher level of abstraction, the presence of some specific code smells can reveal some antipatterns [Brown *et al.*, 1998], of which code smells are symptoms [Brown *et al.*, 1998 ; Fowler, 1999]. One example of a code smell is the ComplexClass smell, which occurs in classes with a very high McCabe complexity when compared to other classes in a system.

As discussed in Chapter 2, despite the existence of many work on code smells and antipatterns, the only work that attempted to study the relation between code smells and fault-proneness was by Wei and Raed [2007]. No other previous work has contrasted the change-proneness of classes with code smells with that of other classes to study empirically the impact of code smells on this software evolution phenomena. In the following, we study 29 code smells [Brown *et al.*, 1998 ; Fowler, 1999], shown in Table 3.3.

Table 3.3 – List of code smells considered in this study (definitions are presented in Appendix B).

AbstractClass	ChildClass
ClassGlobalVariable	ClassOneMethod
ComplexClassOnly	ControllerClass
DataClass	FewMethods
FieldPrivate	FieldPublic
FunctionClass	HasChildren
LargeClass	LargeClassOnly
LongMethod	LongParameterListClass
LowCohesionOnly	ManyAttributes
MessageChainsClass	MethodNoParameter
MultipleInterface	NoInheritance
NoPolymorphism	NotAbstract
NotComplex	OneChildClass
ParentClassProvidesProtected	RareOverriding
TwoInheritance	

3.2.1 Study Definition and Design

Context: We analyze the change history of two systems, Azureus and Eclipse, having different sizes and belonging to different domains. Azureus[2], now known as Vuze, is an open source BitTorrent client written in Java. BitTorrent is a protocol that allows to exchange files over the Internet. Eclipse[3] is an open-source integrated development environment used both in open-source communities and in industry. It is mostly written in Java, with C/C++ code used mainly for widget toolkits. It is also developed partly by a commercial company, IBM, and thus is more likely to embody industrial practices. It

[2] `http://azureus.sourceforge.net/`
[3] `http://www.eclipse.org/`

Table 3.4 – Summary of the 9 releases of Azureus (changes are counted from one release to the next, Azureus 4.2.0.2 is thus excluded).

Dates	Releases	Numbers of		
		LOCs	Classes	Changes
2008-06-16	3.1.0.0	589,049	2,954	669
2008-07-01	3.1.1.0	604,527	3,026	7,035
2008-10-15	4.0.0.0	690,116	3,045	383
2008-10-24	4.0.0.2	648,942	3,099	387
2008-11-20	4.0.0.4	651,642	3,111	1,589
2009-01-26	4.1.0.0	664,163	3,149	238
2009-02-05	4.1.0.2	664,554	3,149	478
2009-02-25	4.1.0.4	664,810	3,150	1,341
2009-03-23	4.2.0.0	680,238	3,210	106
Total	9	5,858,041	27,893	12,226

has been used by other researchers in related studies, *eg.*, to predict faults [Zimmermann *et al.*, 2007].

We analysed 9 releases of Azureus, from release 3.1.0.0 to 4.2.0.0, in the years 2008-2009. Characteristics of the analysed releases are shown in Table 3.4. We analysed 13 releases of Eclipse available on the Internet between 2001 and 2008. Table 3.5 summarises the analysed releases and their key figures.

Research Questions: Based on the data collected from Azureus and Eclipse, our study aimed at answering three research questions on the relationship between code smells and classes change-proneness,

- **RQ1:** *What is the relation between code smells and class change proneness?* We investigate whether classes with code smells are more change-prone than others by testing the null hypothesis: H_{01}*: the proportion of classes undergoing at least one change between two releases does not significantly differ between classes with code smells and other classes.*

- **RQ2:** *What is the relation between the number of code smells in a class and its change-proneness?* We are also interested to evaluate whether classes with a higher number of code smells are more change-prone than others by testing the null hypothesis: H_{02}*: the number of code smells in change-prone classes is not significantly higher than the number of code smells in classes that do not change.*

Table 3.5 – Summary of the 13 analysed releases of Eclipse (changes are counted from one release to the next, Eclipse 3.4 is thus excluded).

Dates	Releases	Numbers of		
		LOCs	Classes	Changes
2001-11-07	1.0	781,480	4,647	21,553
2002-06-27	2.0	1,249,840	6,742	26,378
2003-06-27	2.1.1	1,797,917	8,730	10,397
2003-11-03	2.1.2	1,799,037	8,732	11,534
2004-03-10	2.1.3	1,799,702	8,736	15,560
2004-06-25	3.0	2,260,165	11,166	11,582
2004-09-16	3.0.1	2,268,058	11,192	24,150
2005-03-11	3.0.2	2,272,852	11,252	49,758
2006-06-29	3.2	3,271,516	15,153	2,745
2006-09-21	3.2.1	3,284,732	15,176	11,854
2007-02-12	3.2.2	3,286,300	15,184	10,682
2007-06-25	3.3	3,752,212	17,162	7,386
2007-09-21	3.3.1	3,756,164	17,167	40,314
Total	13	31,579,975	151,039	243,903

- **RQ3:** *What is the relation between particular kinds of code smells and change proneness?* Also, we analyse whether particular kinds of code smells contribute more than others to changes by testing the null hypothesis: H_{03}*: classes with particular kinds of code smells are not significantly more change-prone than other classes.*

Variable Selection: We relate the following dependent and independent variables to test the previous null hypotheses and, thus, answer the associated research questions.

- Independent variables: We have as many independent variables as kinds of code smells; we investigate the presence of 29 different kinds of code smells. Each variable $s_{i,j,k}$ indicates the number of times a class i has a smell j in a release r_k. For RQ1, we aggregate these variables into a Boolean variable $S_{i,k}$ indicating whether a class i has at least one smell of any kind. For RQ2, we consider the number of changes $c_{i,k}$ a class i to underwent between r_k and r_{k+1}, and convert $c_{i,k}$ into a Boolean variable $C_{i,k}$ (*true* if the class underwent at least one change, *false* otherwise).

- Dependent variables: The dependent variables measure the phenomena related to our independent variables. Our dependent variable for RQ1 and RQ3 is the class *change proneness*, which is measured, as the number of changes $c_{i,k}$ that a class i underwent between release r_k (in which it has some code smells) and the subsequent release r_{k+1}. For RQ1 and RQ3, we are interested to distinguish classes that underwent,

between two releases, at least one change. In RQ2, we compare the number of code smells in change-prone classes with that in non-change-prone classes, using as dependent variable the total number of code smells $st_{i,k}$ a class i has in a release r_k.

Analysis Method:

- In RQ1, to attempt rejecting H_{01}, we test whether the proportion of classes exhibiting (or not) at least one change, significantly varies between classes with (some) code smells and other classes. We use Fisher's exact test presented in Chapter 4 and compute odds ratio (OR).
- In RQ2, we use the (non parametric) Wilcoxon test (see Chapter 4) to compare the number of code smells in change-prone classes with the number of code smells in non-change-prone classes. We also test the hypothesis with the (parametric) t-test. Other than testing the hypothesis, it is of practical interest to estimate the magnitude of the difference of the number of code smells in classes with and without changes: we use the Cohen d effect size (see [Sheskin, 2007] and Chapter 4) to estimate this magnitude.
- In RQ3, we use a logistic regression model (see [Hosmer and Lemeshow, 2000] and Chapter 4) to relate change-proneness with the presence of particular kinds of code smells. While in other contexts (*eg.*, [Gyimóthy *et al.*, 2005]), logistic regression models were used for prediction purposes; as in [Vokac, 2004], we use logistic regression to reject H_{03}. Then, for each smell and for the 9 analysed Azureus releases and for the 13 Eclipse releases, we count the number of times that the p-values obtained by the logistic regression are significant. During the procedure for building the logistic regression model, we discard variables that are highly correlated to others—that can happen between some code smells—thus the model only contains a non-redundant set of code smells useful to warn against classes change-proneness.

3.2.2 Study Results

We now report the results of our study to address the three research questions.

3.2.2.1 RQ1: Smells and Change Proneness

Tables 3.6 and 3.7 report, for each analysed release of Azureus and Eclipse, the number of classes (1) with code smells and that changed; (2) with code smells but that did not

Table 3.6 – Azureus: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
3.1.0.0	220	1967	20	1433	$<$ **0.01**	8.01
3.1.1.0	564	1686	101	1381	$<$ **0.01**	4.57
4.0.0.0	83	2238	7	1519	$<$ **0.01**	8.05
4.0.0.2	106	2206	12	1510	$<$ **0.01**	6.04
4.0.0.4	435	1886	39	1484	$<$ **0.01**	8.77
4.1.0.0	50	2297	11	1533	$<$ **0.01**	3.03
4.1.0.2	112	2235	11	1533	$<$ **0.01**	6.98
4.1.0.4	112	2236	12	1532	$<$ **0.01**	6.39
4.2.0.0	37	2353	3	1580	$<$ **0.01**	8.28

change; (3) without code smells but with changes; and, (4) with neither code smells nor changes. The tables also report the result of Fisher's exact test and ORs when testing H_{01}.

Results for Azureus in Table 3.6 show that the proportions are significantly different, therefore, we reject H_{01}. Moreover, ORs are very high (always greater than 3); in most cases the odds for classes with code smells to change is six times higher or more than for classes without code smells. H_{01} rejection and the ORs provide *a posteriori* concrete evidence of the negative impact of code smells on change-proneness. Developers should be wary of classes with code smells, because they are more likely to be the subject of their maintenance effort. Although changes are not necessary bad, too frequent changes are costly.

For Eclipse, except for the 3.0 release series, proportions are significantly different, thus allowing to reject H_{01}. There is a greater proportion of classes with code smells that change with respect to other classes. In some cases (*eg.*, releases 1.0, 2.0, 2.1.2, 2.1.3, and the 3.0 release series), ORs are close to 1, *i.e.*, the odds is even that a class with a code smell changes or not. In the other releases, the odds of changing are 2 to 3.6 times in favour of classes with code smells.

We conclude that the odds to change are in general higher for classes with code smells. Code smells classes are therefore likely to be more change-prone and therefore less maintainable.

Table 3.7 – Eclipse: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
1.0	2042	1731	417	448	$<$ **0.01**	1.27
2.0	3673	1373	767	236	**0.02**	0.82
2.1.1	2224	3838	193	964	$<$ **0.01**	2.89
2.1.2	2400	3664	359	798	$<$ **0.01**	1.46
2.1.3	2942	3125	516	642	**0.01**	1.17
3.0	3415	4880	684	1032	0.32	1.06
3.0.1	6216	2087	1294	423	0.69	0.97
3.0.2	5784	2520	1194	524	0.91	1.01
3.2	1819	9621	115	2210	$<$ **0.01**	3.63
3.2.1	2778	8680	291	2038	$<$ **0.01**	2.24
3.2.2	3321	8144	409	1921	$<$ **0.01**	1.92
3.3	1778	10844	145	2364	$<$ **0.01**	2.67
3.3.1	4337	8290	682	1830	$<$ **0.01**	1.40

3.2.2.2 RQ2: Number of Smells and Change Proneness

Tables 3.8 and 3.9 report, for Azureus and Eclipse respectively, results of the Wilcoxon two-tailed test, t-test, and Cohen d effect size, aimed at comparing the number of code smells in classes that changed or not. For Azureus, the p-values are always significant with a high effect size, indicating that; for all analysed releases, change-prone classes are those with a higher number of code smells. For Eclipse, results are significant (although with a small effect size), except for the 3.0 release series, where differences are not significant, thus confirming the findings from RQ1 regarding the limited relation of code smells with change-proneness for this release series. In summary we can reject H_{02} , *i.e.*, classes with a higher number of code smells are more change-prone than others.

3.2.2.3 RQ3: Kinds of Smells and Change Proneness

Tables 3.10 and 3.11 show the results of the logistic regression for the correlations between changes and the different kinds of code smells. The tables summarise the number of analysed releases in which each kind of code smells was significant in the logistic regression model. Code smells that are significant for at least 75% of the releases (7 for Azureus, 10 for Eclipse) are highlighted in boldface. In Azureus, only the code smell NotAbstract

Table 3.8 – Azureus: Wilcoxon and t-test results for number of code smells in classes that are change-prone or not.

Releases	Wilcoxon p	t-test p	Cohen d
3.1.0.0	$<$ **0.01**	$<$ **0.01**	0.72
3.1.1.0	$<$ **0.01**	$<$ **0.01**	0.71
4.0.0.0	$<$ **0.01**	$<$ **0.01**	1.01
4.0.0.2	$<$ **0.01**	$<$ **0.01**	0.86
4.0.0.4	$<$ **0.01**	$<$ **0.01**	0.83
4.1.0.0	$<$ **0.01**	$<$ **0.01**	0.59
4.1.0.2	$<$ **0.01**	$<$ **0.01**	0.93
4.1.0.4	$<$ **0.01**	$<$ **0.01**	0.85
4.2.0.0	$<$ **0.01**	$<$ **0.01**	1.02

Table 3.9 – Eclipse: Wilcoxon and t-test results for number of code smells in classes that are change-prone or not.

Releases	Wilcoxon p	t-test p	Cohen d
1.0	0.79	**0.03**	0.06
2.0	$<$ **0.01**	$<$ **0.01**	−0.08
2.1.1	$<$ **0.01**	$<$ **0.01**	0.31
2.1.2	$<$ **0.01**	$<$ **0.01**	0.13
2.1.3	**0.04**	$<$ **0.01**	0.07
3.0	0.07	0.10	0.03
3.0.1	0.11	0.26	−0.03
3.0.2	0.12	0.28	−0.02
3.2	$<$ **0.01**	$<$ **0.01**	0.41
3.2.1	$<$ **0.01**	$<$ **0.01**	0.29
3.2.2	$<$ **0.01**	$<$ **0.01**	0.25
3.3	$<$ **0.01**	$<$ **0.01**	0.41
3.3.1	$<$ **0.01**	$<$ **0.01**	0.18

has a significant impact on change proneness in more than 75% of releases. AbstractClass and LargeClass are significant in more than 50% of the releases (5 out of 9). In Eclipse, the code smells that have a significant effect on change-proneness for 75% of the releases or more are HasChildren, MessageChainsClass, and NotComplex. In summary, although results sometimes depend on the particular context—*eg.*, system analysed and particular release—we can reject H_{03}, *i.e.*, there are code smells that are more related than others to change-proneness.

3.2.3 Summary

In this section, we reported an exploratory study, performed on 9 releases of Azureus and 13 releases of Eclipse, which provides empirical evidence of the negative impact of

Table 3.10 – Azureus: number of significant p-values in the 9 analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of code smells. Boldface and gray background indicate significant p-values for at least 75% of the releases.

Smells	Proneness to Changes
AbstractClass	5
ChildClass	3
ClassGlobalVariable	2
ClassOneMethod	1
ComplexClassOnly	2
ControllerClass	2
DataClass	4
FewMethods	2
FieldPrivate	1
FieldPublic	2
FunctionClass	2
HasChildren	1
LargeClass	5
LargeClassOnly	–
LongMethod	–
LongParameterListClass	1
LowCohesionOnly	2
ManyAttributes	–
MessageChainsClass	4
MethodNoParameter	2
MultipleInterface	4
NoInheritance	3
NoPolymorphism	3
NotAbstract	**7**
NotComplex	2
OneChildClass	1
ParentClassProvidesProtected	–
RareOverriding	1
TwoInheritance	–

code smells on classes change-proneness. We showed that classes with code smells are significantly more likely to be subject to changes than other classes. We also showed that some specific code smells are more likely to be of concern during evolution. Building on these results, we perform and report in Chapter 6 a larger study on antipatterns and classes change- and fault- proneness.

3.3 Pilot Study 3: Do Issues Reporting Systems Report Faults?

Issues Reporting Systems (IRS) are valuable assets for managing maintenance activities. They are widely used in open-source projects as well as in the software industry. Although they should be used mostly to manage issues related to corrective maintenance,

Table 3.11 – Eclipse: number of significant p-values in the 13 analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of code smells. Boldface and gray background indicate significant p-values for at least 75% of the releases.

Smells	Proneness to Changes
AbstractClass	1
ChildClass	6
ClassGlobalVariable	2
ClassOneMethod	4
ComplexClassOnly	8
ControllerClass	4
DataClass	4
FewMethods	2
FieldPrivate	6
FieldPublic	8
FunctionClass	1
HasChildren	**11**
LargeClass	8
LargeClassOnly	–
LongMethod	9
LongParameterListClass	6
LowCohesionOnly	5
ManyAttributes	9
MessageChainsClass	**10**
MethodNoParameter	8
MultipleInterface	5
NoInheritance	–
NoPolymorphism	3
NotAbstract	1
NotComplex	**10**
OneChildClass	2
ParentClassProvidesProtected	–
RareOverriding	4
TwoInheritance	–

they happen to collect many other kinds of issues: requests for enhancements, refactoring/restructuring activities, and organizational issues. These different kinds of issues are simply labeled as “bug” for lack of a better classification or of knowledge about the possible kinds.

In this pilot study, we analysed data contained in IRS of three open source systems, Eclipse, JBoss, and Mozilla, to understand to which extent these data are related to corrective maintenance issues.

3.3.1 Study Definition and Design

Context: The context of this study consists in the IRS of the three well-known, industrial-strength, open-source systems[4].

Eclipse is an open-source integrated development environment (see Section 3.2.1 for a description). We mirrored locally the CVS and bug repositories of Eclipse at the end of 2006 and extracted all the bugs. Then, we selected 10,386 bugs which were tagged as either "Verified" or "Resolved", *i.e.*, bugs for which a resolution is known.

The Mozilla suite is an open-source suite implementing a Web browser and other tools, such as a mailer and a newsreader. It was ported on almost all software and hardware platforms. It is developed mostly in C++, with C code accounting for only a small fraction of the system. As for Eclipse, we are interested in the 92,858 bugs that are tagged as "Verified" or "Resolved".

JBoss is an enterprise-application platform and Web-service application stack to develop, deploy, and manage Java service-oriented enterprise applications. It is almost entirely developed in Java and XML plus shell scripts and batch files. As for Mozilla and Eclipse, we concentrated our effort on bugs for which a resolution was known. JBoss bugs are store in Jira. We use a RSS feeder to extract the 3,207 issues classified as "Resolved".

In this study, we chose to select issues with the "Resolved" or "Closed" status to avoid duplicated bugs, rejected issues, or issues awaiting triage.

Research Questions: Our study aims at answering the following research question:

- **RQ1:** *To what extent the information contained in issues posted on IRS is related to faults?*

Analysis Method: We proceed in two steps to answer **RQ1**:

1. We validate manually 1,800 randomly-selected issues from the three IRSs and compute the proportion of fault-related issues.
2. We use the 1,800 manually validated issues as oracle and build classifiers. With these classifiers, we automatically classify the remaining issues and estimate the overall proportion of fault-related issues.

[4] `http://www.eclipse.org/`, `http://www.mozilla.org/`, and `http://www.jboss.org/`

Classification: The magnitude of the numbers of issues in these systems makes it clearly infeasible to manually classify each of these issues as either corrective maintenance (faults) or not. Therefore, instead of manually validating all the issues in the three IRS, we randomly sampled and manually classified 600 issues for each system. Overall, we classified 1,800 distinct issues.

We organised the issues in bundles of 150 entries each. For every subset, we asked three developers to classify the issues manually. They were asked to state if the issues were a corrective maintenance (hereby referred to as "fault") or a non-corrective maintenance (enhancement, refactoring, re-documentation, or other, *i.e.*, "non-fault"). The classifications went through a simple majority vote. An issue was considered a corrective maintenance if at least two out of the three developers classified it as a corrective maintenance (*i.e.*, "fault"). Otherwise the issue was considered as a non-corrective maintenance (*i.e.*, "non-fault").

We used these manually classified issues as oracle. We processed them to extract their characteristics that were used as independent variables in our various supervised techniques.

Specifically, we mapped each IRS issue into a vector of raw frequencies of terms appearing in the issue's description; example of terms are: "failure", "crash", or "should".

Next, we augmented each issue from the oracle with its class $\{0, 1\}$, *i.e.*, $\{non\ fault, fault\}$. This column was used by the machine learning techniques during the training phase.

Finally, we performed the automatic classification of IRS issues using the *Weka* tool[5], in particular we used the symmetrical uncertainty attribute selector, the standard probabilistic naive Bayes classifier, the alternating decision tree (ADTree), and the linear logistic regression classifier. Our choice was motivated by the observation that these machine learning techniques produce classifiers more easily interpretable. We present and discuss these techniques in Chapter 4.

3.3.2 Study Results

Table 3.12 reports the results of the manual classification. The last column of the table reports issues that have nothing to do with fault fixing or evolution: *eg.*, user complains of an incompatibility with a version of an operating system library, requests of an obsolete

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

Table 3.12 – Manual classification of the 1,800 IRS issues for Mozilla, Eclipse and JBoss.

Systems	Faults	Non Faults	Others
Mozilla	270	209	121
Eclipse	194	382	24
JBoss	345	99	156

release, requests of fault fixing of a component not belonging to the system, requests for write access to SVN/CVS repository, configuration help, and so on.

The results suggest that Eclipse, JBoss, and Mozilla IRS contain a large fraction of non-fault issues; only 44.9% of the 1,800 issues are faults.

We also found that the information contained in issues posted on IRS can be indeed used to automatically classify such issues, distinguishing faults from other activities, with a precision between 64% and 98% and a recall between 33% and 97% and a correct decision rate as high as 82%.

The automatic classification of the remaining issues confirmed that less than half are relate to corrective maintenance, *i.e.*, faults.

3.3.3 Summary

This study shows that IRS, in open-source development, have a far more complex use than simple bookkeeping of corrective maintenance and studies based on IRS issues should carefully consider which issues are used to build their predictive models. Out of our 1,800 manually-classified issues, we found only 44.9% to be related to faults. Globally, less than half of the issues posted on IRS are faults. For our experiments in the Chapter 6 of this dissertation, we will rely as much as possible on manually-validated data. We will use a set of manually-validated and publicly-available faults for Mylyn and Rhino [Eaddy *et al.*, 2008]. This choice reduces biases found in previous models from the literature (see [Bird *et al.*, 2009]).

3.4 Conclusion

This chapter presented the results of three preliminary studies on design patterns, code smells, and quality of data. In the first study, we surveyed developers to assess their perception of the impact of the 23 design patterns from [Gamma *et al.*, 1994] on ten quality

attributes. Contrary to common lore, the results suggested that although design patterns impact the quality of systems, they do not always improve this quality. Some patterns are considered by developers to decrease certain quality attributes and to not necessarily promote reusability, expandability, and understandability as claimed in [Gamma *et al.*, 1994]. Building on these results, we perform and report in Chapter 5 a detail analysis of design patterns implementations and their impact on the quality of systems.

The second study concerned the change proneness of code smells. We studied 9 releases of Azureus and 13 releases of Eclipse and found that classes with smells are significantly more likely to be subject to changes than other classes; with specific code smells like MessageChainClass, NotAbstract, and HasChildren more likely to be of concern during evolution. This result provides empirical evidence of the negative impact of code smells on classes change-proneness. In Chapter 6, we extend this study and report a larger study on antipatterns and classes change- and fault- proneness.

The third study investigated data contained in the IRS of three open source systems, Eclipse, JBoss, and Mozilla to understand the extend to which these data are related to corrective maintenance issues. From these IRS, we manually-classified 1,800 randomly-selected issues and automatically classified the others. We found less than half of these issues to be related to corrective maintenance. Consequently, we chose to use a set of manually-validated and publicly-available faults for Mylyn and Rhino [Eaddy *et al.*, 2008] for the experiments reported in this dissertation.

Chapter 4

Experimental Settings

This chapter presents the techniques and tools used in the following experiments. First, we present a brief review of the literature on design patterns and antipatterns identification techniques followed by a description of the tools used in this dissertation. We also describe the variables and the statistical methods used in our experiments.

4.1 Design Motif Identification

This section introduces and discusses the techniques and tools used to identify classes playing roles in design motifs (a design motif is a concrete solution of a design patterns [Guéhéneuc and Antoniol, 2008]), historically called design motif identification techniques.

The first tool to identify design motifs in systems was proposed by Brown [1996]. He reverse-engineered Smalltalk code to identify the Composite, Decorator, Template Method, and Chain of Responsibility design motifs from the catalog [Gamma *et al.*, 1994]. He introduced KT, a design motif identification tool based on information retrieved from class hierarchies; association and aggregation relationships, as well as the messages exchanged between classes of a system.

Following Brown's work, Krämer and Prechelt [1996] introduced the Pat system to identify occurrences of design motifs in C++ source code. In their work, design motifs from a library are expressed as Prolog predicates and systems (extracted from C++ headers) as Prolog facts. They reported a precision of 40% on four C++ systems and five design motifs.

Shull *et al.* [1996] introduced the first attempt to identify recurring design motifs in a system without resorting to a pre-defined library of motifs. They proposed a manual method, BACKDOOR (Backwards Architecting Concerned with Knowledge Discovery of Object Oriented Relationships), to infer systematically occurrences of motifs in a system. They illustrated their method on seven student systems. Tonella and Antoniol [1999] used concept analysis to also identify occurrences of recurring motifs without a pre-defined library. They were able to retrieve many occurrences of the Adapter design motif.

Other techniques introduced for design motif identification include the framework SOUL for Smalltalk [Wuyts, 1998]. This framework allows a direct representation of the abstract syntax tree of the Smalltalk source code as logic facts. Using these facts, it is possible to build a repository of predicates to identify classes whose structures and organizations correspond to design motifs.

Antoniol *et al.* [1998 ; 2001] presented a systematic approach to recover design motifs from design and code. Their approach is based on a multi-stage reduction strategy performed using metrics and meant to avoid combinatorial explosion on large systems. It maps code and design into an intermediate representation, called Abstract Object Language (AOL), and represents a motif as a tuple of classes and relations among classes. It uses metrics and methods calls to determine motifs constituent's candidate sets.

Seeman and von Gudenberg [1998] proposed an approach to identify occurrences of design motifs using data collected from a compiler, including inheritance hierarchies, method invocations, class names, and method names. These data are used to abstract the system as a graph. From this graph, binary classes relationships are inferred to build a new graph in which design motifs are inferred based on a library of pre-defined motifs using first order logic. Philippow *et al.* [2005] proposed an identification algorithm for occurrences of design motifs based on tailored heuristics. Their heuristics used negative data about the relationships and characteristics of classes playing roles in design motifs to improve precision. They provided the complexity of their algorithms on many design motifs and concluded on the need for user interaction during their identification process.

Niere *et al.* [2001 ; 2002] proposed the use of fuzzy set theory to handle variant implementations of design motifs when searching for their occurrences. They described an approach for the identification of occurrences of design motifs in systems handling complete and incomplete occurrences. In their work, they divided motifs into sub-motifs, for example association. They implemented their approach in Fujaba.

Keller *et al.* [1999] proposed the SPOOL environment for the recovery of components and design rationale in software systems. At the heart of the SPOOL environment is

a UML meta-model to describe various models of a system. These models are reverse-engineered from C++, Java, or Smalltalk source code through a UML/CDIF parser. With their environment, motif identification can be manual, semi-automatic, or full-automatic, based on stored abstract models of design motifs. SPOOL also includes various visualisation techniques. A validation was performed on two industrial systems and ET++ using the Template Method, Factory Method, and Bridge design motifs. Only numbers of occurrences are reported.

Tsantalis *et al.* [2006] implements an approach based on similarity scoring between graph vertices to identify design motifs in systems. The tool takes as inputs both the system and the motif graph and computes similarity scores between their vertices. Due to the nature of its underlying graph algorithm, this approach is able to recognize not only the motifs described in [Gamma *et al.*, 1994] but also variants. The approach uses an implicit notion of main roles (roles with the most unique characteristics) and do not consider roles played by artifacts other than classes or interfaces (for example methods). Therefore, the approach fails to distinguish between motifs like Adapter and Command because the distinction between them would require data on their methods. The tool analyses Java bytecode and identifies the two main participants of each motif. The tool is able to detect the following motifs: Object Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, Template Method, and Visitor. In it detection results, it reports the main participants of classes belonging to a motif and their descendants. A validation of the tool on three open-source systems (JHotDraw, JRefactory, and JUnit) reported few false negatives and no false positives.

Kaczor *et al.* [2009] proposed a design motifs identification algorithm based on bit-vectors. Design motifs and systems are converted into sequences of characters by converting their graphs into Eulerian graph and traversing these graphs using the Chinese postman algorithm. They illustrated their algorithm using the Composite design motifs and reported 100% recall on three systems.

Some design motifs identification approaches use dynamic analysis. In this category, Ng *et al.* [2009] proposed to identify occurrences of behavioural and creational design motifs using dynamic analysis. They described both motifs and systems in the form of scenario diagrams (subsets of sequence diagrams) and used explanation-based constraint programming to identify complete and incomplete occurrences. They reported results for Memento and Visitor motifs in JHotDraw that showed good precision, recall, and performance.

In this dissertation, we use the PTIDEJ framework (Pattern Trace Identification, Detection, and Enhancement in Java) by Guéhéneuc [2005 ; 2009]. The PTIDEJ framework implements the DeMIMA detection approach by Guéhéneuc and Antoniol [2008] and the approaches by Kaczor *et al.* [2009] and Ng *et al.* [2009] presented above. Combining these approaches, PTIDEJ is able to detect 13 design motifs, namely: Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method, and Visitor.

DeMIMA [Guéhéneuc and Antoniol, 2008], a multilayered approach for design motif identification, consists of three layers. The first two layers are for the recovery of an abstract model of the source code, including binary class relationships. A third layer is intended for the identification of design motifs in the abstract model. The approach uses explanation-based constraint programming to automatically provide (1) explanations on the identified occurrences: the roles and relationships that led to identify a certain micro-architecture as an occurrence of a motif and (2) approximations from the given motifs: the relaxations of the constraints needed for a micro-architecture to be identified as an approximated occurrence of a motif. It provides a complete mapping between roles in a motif and classes in an occurrence. Thus, with this approach, it is possible to find all the roles that each class plays for a set of motifs in a system. DeMIMA ensures a recall of 100% by automatically relaxing appropriate constraints and precision is between 34% and 80% on average [Guéhéneuc and Antoniol, 2008].

4.2 Antipatterns Detection

Webster [1995] wrote the first book on "antipatterns" in object-oriented development; his contribution covers conceptual, political, coding, and quality-assurance problems. Riel [1996] defined 61 heuristics characterising good object-oriented programming to assess software quality manually and improve design and implementation. Fowler [1999] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [2003] and Wake [2003] proposed classifications of code smells. Brown *et al.* [1998] described 40 antipatterns, including the well-known Blob. These books provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide academic and industrial audience.

Several approaches to specify and identify occurrences of code smells and antipatterns have been proposed in the literature. They range from manual approaches, based on inspection techniques [Travassos *et al.*, 1999], to metric-based heuristics [Marinescu, 2004 ; Munro, 2005 ; Moha *et al.*, 2009], where antipatterns are identified according to sets of rules

and thresholds defined on various metrics. Manual approaches were defined, for example, by Travassos *et al.* [1999], who introduced manual inspections and reading techniques to identify code smells.

Marinescu [2004] presented a metric-based approach to identify code smells with detection strategies, which captures deviations from good design principles and consists of combining metrics with set operators and comparing their values against absolute and relative thresholds. Similarly to Marinescu, Munro [2005] proposed metric-based heuristics to identify code smells; the heuristics are derived from a template similar to the one used for design motifs. He also performed an empirical study to justify the choice of metrics and thresholds for detecting code smells.

Rao and Reddy [2008] proposed the use of design change propagation probability matrix to detect two bad smells: Shotgun Surgery and Divergent Change. In their identification process, they quantified and used design change propagation between classes that are connected directly and indirectly. They claimed that this quantification was essential for an effective detection of the smells. They applied their method on three small size systems and suggested, for the two studied smells, refactorings to improve the quality of designs. They introduced a framework implementing the approach.

Some visualisation techniques, for example [Simon *et al.*, 2001], were used to find a compromise between fully-automatic identification techniques, which are efficient but lose track of the context, and manual inspections, which are slow and subjective. Other approaches perform fully-automatic identification and use visualisation to present the identification results [van Emden and Moonen, 2002 ; Lanza and Marinescu, 2006].

In this dissertation, we use our approach, BDTEX [Khomh *et al.*, 2009b], and DECOR (Defect dEtection for CORrection) by Moha *et al.* [2009], to specify and identify antipatterns and code smells. DECOR is a detection approach based on a thorough domain analysis of code smells and antipatterns in the literature; analysis on which is based a domain-specific language. DECOR also proposes algorithms and a platform to automatically convert rule cards into detection algorithms [Moha *et al.*, 2009]. DECOR can be applied on any object-oriented system because it is based on PADL and POM, described in details in Section 4.3.

Moha *et al.* [2009] reported that DECOR current detection algorithms for antipatterns ensure 100% recall and have precisions between 41.1% and 87% for three antipatterns: Blob, SpaghettiCode, and SwissArmyKnife. The detection algorithms for these three antipatterns have an average accuracy of 99% for the Blob, of 89% for the SpaghettiCode, and of 95% for the SwissArmyKnife; and a total average of 94%.

DECOR is able to detect the following antipatterns from [Brown *et al.*, 1998 ; Fowler, 1999]: AntiSingleton, Blob, ClassDataShouldBePrivate (CDSBP), ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList (LPL), MessageChains, RefusedParentBequest (RPB), SpaghettiCode, SpeculativeGenerality (SG), SwissArmyKnife. These antipatterns are representative of design and implementation problems with data, complexity, size, and the features provided by a class.

In addition to antipatterns detection algorithms, DECOR also provides code smells detection algorithms (antipatterns are defined in terms of code smells; however, some code smells are also considered antipatterns) with a higher precision (80% on average) for the following code smells: AbstractClass, ChildClass, ClassGlobalVariable, ClassOneMethod, ComplexClassOnly, ControllerClass, DataClass, FewMethods, FieldPrivate, FieldPublic, FunctionClass, HasChildren, LargeClass, LargeClassOnly, LongMethod, LongParameterListClass, LowCohesionOnly, ManyAttributes, MessageChainsClass, MethodNoParameter, MultipleInterface, NoInheritance, NoPolymorphism, NotAbstract, NotComplex, OneChildClass, ParentClassProvidesProtected, RareOverriding, TwoInheritance.

However, threshold-based detection techniques such as DECOR do not handle the uncertainty of the detection results [Dhambri *et al.*, 2008 ; Oliveto *et al.*, 2010] and, therefore, miss borderline classes, *i.e.*, classes with characteristics of antipatterns "surfacing" slightly above or "sinking" slightly below the thresholds because of minor variations in the characteristics of these classes. In a previous work [Khomh *et al.*, 2009b], we proposed BDTEX, an antipattern identification approach using BBNs to address this issue. BDTEX is a GQM-based approach to build Bayesian Beliefs Networks from the definitions of antipatterns. The symptoms specifying antipatterns are selected by a quality analyst, thus ensuring that the BBN is qualitatively sound. The output of the BBN is the probability that a class exhibiting the symptoms of an antipattern is really such an antipattern. These BBNs work with missing data and can be tuned using quality analysts' knowledge. In addition, with BDTEX, candidate classes, *i.e.*, potential antipatterns, are associated with probabilities, which indicate the degree of uncertainty that a class is indeed an occurrence of some antipattern. These probabilities can help focus manual inspection by ranking candidate classes. BDTEX is able to detect the same antipatterns and code smells as DECOR with results globally superior in terms of precision, recall, and quality analysts' utility (*eg.*, on the Blob, BDTEX achieves 57.1% average precision on GanttProject, while DECOR reports 26.73%).

The definition of code smells and the antipatterns studied in this dissertation and discussed in this section are presented in Appendix B.

4.3 Metrics Computation

This section presents POM, the tool we used to compute metrics. POM is a framework that offers more than 60 different metrics from the literature, including class-method import and export coupling [Briand *et al.*, 1997]; Coupling Between Objects (CBO), and Weighted Method Count (WMC) [Chidamber and Kemerer, 1993]; Lack of Cohesion in Methods (LCOM5) [Henderson-Sellers, 1995]; 'C' connectivity of a class [Hitz and Montazeri, 1995]; numbers of new, inherited, and overridden methods and total number of methods [Lorenz and Kidd, 1994]; Cyclomatic Complexity Metric (CC) [McCabe and Butler, 1989]; numbers of hierarchical levels below a class and class-to-leaf depth [Tegarden *et al.*, 1995]. The definitions of all the metrics implemented in POM is presented in in Appendix A.

In addition to metrics, the POM framework offers statistical features for computing and accessing metrics box-plots. POM is based on the meta-model PADL, which provides parsers to describe object-oriented systems written in C♯, C++, and Java [Guéhéneuc and Antoniol, 2008]. With this meta-model, models of systems are built and the metric values are computed by applying each metric on each class of the models.

4.4 Change- and Fault-proneness Computation

Change- and fault-proneness in this dissertation are computed using our Ibdoos framework. Ibdoos extracts commit information from any CVS, GIT, or SVN repository and stores this in a database. We query the database to compute change and fault proneness as follows:

Change-proneness: Change-proneness refers to whether a class underwent *at least a change* between release k (in which it was participating for example in some antipatterns) and the subsequent release $k + 1$. Changes are identified, for each class in a system, by looking at commits in the control-version system (CVS or SVN); for each class, we counted, the number of commits related to that class.

A class is change prone if, at a given time, it has been changed more than other classes.

Fault-proneness: Fault-proneness refers to whether a class underwent *at least a fault-fixing* between releases k and $k + 1$.

To compute the fault-proneness of classes, as discussed in Chapter 3, we consider a set of manually-validated and publicly-available faults for Mylyn and Rhino [Eaddy *et*

al., 2008]. For ArgoUML, issues dealing with fixing faults are marked as "DEFECT" in the issue tracking system[1]. For Eclipse, however, we cannot distinguish issues related to faults from other issues because Bugzilla is used as a general issue-tracking system and does not provide tagged fault-related issues. Given the high number of issues, 34,634 between releases 1.0 and 3.4, a manual classification is not practical. Consequently, while we analyse fault-proneness for ArgoUML, Mylyn, and Rhino, we analyse only issue-proneness for Eclipse, by considering issues marked as "FIXED" or "CLOSED", because they required some changes. We trace faults/issues to changes by matching their IDs in the commits [Fischer *et al.*, 2003].

We are aware that changes and faults are related. Indeed, the more a piece of code changes, the more likely it might have a fault. Therefore, it would be interesting to control for faults when looking at changes and control for changes when looking at faults. We plan to investigate that in future work.

Kinds of changes: Kinds of changes are counted as the number of each kind of change occurring to a class in release k. We determine different kinds of changes performed on a class c_i, by comparing the class revision in releases k and $k+1$. The comparison is made using an existing source code analyser and differencing tool, and aims at identifying (i) structural changes, *i.e.*, addition/removal/change of/to attributes, addition/removal of methods, or changes to the method signatures, and (ii) non-structural changes, *i.e.*, changes in method implementation.

4.5 Objects in the Experiments

We perform our experiments in this dissertation using the following well-known, industrial-strength, open-source systems[2]

ArgoUML is a full-fledged UML modelling tool with code generation and reverse-engineering capabilities. It provides the user with a set of views and tools to model systems

[1] `http://argouml.tigris.org/issues`

[2] `http://argouml.tigris.org/`, `http://azureus.sourceforge.net/`, `http://www.eclipse.org/`, `http://www.jhotdraw.org/`, `http://xml.apache.org/xalan-j/`, `http://xerces.apache.org/xerces-j/`, `http://www.eclipse.org/mylyn/`, `http://www.eclipse.org/jdt/`, and `http://www.mozilla.org/rhino/`.

using UML diagrams, to generate the corresponding code skeletons and to reverse-engineer diagrams from existing code.

Azureus (now called Vuze) is a bit-torrent client. Bit torrent is a protocol to exchange data among peers across a network. Azureus provides advanced user-interface and implementation of the protocol.

Eclipse is an open-source integrated development environment. It is a platform used both in open-source communities and in industry.

JHotDraw is a graphic framework for drawing 2D graphics. It was created in October 2000 by Beck and Gamma with the purpose of illustrating the use of design patterns.

Xalan is an XSLT processor for transforming XML documents into other document types (HTML, text, and so on). It implements the XSLT and XPath standards.

Xerces is a Java XML parser which supports XML, DOM, and SAX.

Mylyn is a plug-in for Eclipse, which aims at reducing information overload and making developers' multi-tasking easier.

JDT Core is an Eclipse plug-in that implements the infrastructure for the Java IDE of the Eclipse platform. It provides a Java model and capabilities to parse, manipulate, and rewrite Java systems.

Rhino is an open-source implementation of a JavaScript interpreter.

Having presented the tools and objects used in this dissertation, we now provide background information about the statistical methods used in our experiments.

4.6 Background on Statistical Techniques

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

4.6.1 Fisher's Exact Test

The Fisher's exact test [Sheskin, 2007] is a statistical test designed to determine if there are non-random associations between two categorical variables. This test checks whether a proportion varies between two samples by testing the independence of rows and columns in a 2×2 contingency table based on the exact sampling distribution of the observed frequencies. Hence it is an "exact" test.

In our experiments, we also compute *odds ratio* (OR) [Sheskin, 2007] to complement the information of the Fisher's exact test. OR indicates the likelihood of an event to occur. It is defined as the ratio of the odds p of an event occurring in one sample, *eg.*, the set of classes participating in some antipatterns (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the set of classes participating in no antipattern (control group): $\text{OR} = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (antipatterns) while an OR < 1 indicates the opposite (control group).

4.6.2 Wilcoxon Rank-sum Test

The Wilcoxon rank-sum test, also called Mann-Whitney test, is a non-parametric statistical hypothesis test that assesses whether two samples come from a same distribution. It allows us to attempt rejecting null hypotheses while making no assumptions on the normality of the samples. In practice, the t-test is used to complement it.

4.6.3 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 magnitude of a phenomenon under study (*eg.*, the difference of the number of antipatterns in classes with and without changes). This magnitude, called effect size complements p-values and measures the strength of the relationship between two variables (*eg.*, antipatterns and changes). In our experiments, we use the Cohen d effect size.

4.6.4 Cohen d effect size

For independent samples and unpaired analyses, the Cohen d effect size [Sheskin, 2007] is the difference between the means M_1 and M_2 divided by the pooled standard deviation $\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$ of both groups: $d = (M_1 - M_2)/\sigma$. The effect size is small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ [Cohen, 1988].

4.6.5 Machine Learning Techniques

A *classifier* is a function $f\colon R^d \mapsto \text{C}$ that assign a label from a finite set of classes $\text{C} = \{c_1, \ldots, c_q\}$ to observations $\mathbf{x} = (a_1, \ldots, a_d) \in R^d$. In this dissertation, we are interested only in the family of binary classifiers where there are two classes and thus C contains

only two symbols, *eg.*, $C = \{0, 1\}$ or $C = \{non\ fault, fault\}$. We chose these classifiers because they are more easily interpretable.

Three families of machine learning techniques are available to build a classifier: unsupervised learning, supervised learning, and reinforcement learning [Mitchell, 1997 ; Aplaydin, 2004]. Unsupervised learning, for example clustering algorithms, classifies available data based on some fitness or cost function: often a distance or similarity. Supervised learning, *eg.*, ClAssification and Regression Trees (CART), assumes that a training set of labeled data is available. A classifier is then built by maximizing some gain or minimizing a cost function, representative of the accuracy of the classifier with respect to the a-priori classification. In reinforcement learning, a user is required to decide if the classification for the current piece of data is correct; the classifier then incrementally learns a classification function.

Unsupervised learning techniques are appealing because no pre-labeled corpus is needed; however, it is difficult to interpret the resulting classification and hard to derive guidelines linking the classification with characteristics of the data or of the development process.

Supervised learning techniques, in particular algorithms such as CART, Bayesian classifiers, or logistic regression, produce classifiers more easily interpretable than those from unsupervised learning techniques but require a labeled corpus. A labeled corpus is a set of pairs (observation, label) assumed to be random variables $(\mathbf{X}, Y)$ drawn from a fixed but unknown probability distribution μ. The objective of the learning techniques is to find a classifier f with a low error probability $\Pr_\mu[f(\mathbf{X}) \neq Y]$.

Both the selection and the evaluation of f must be based on some data set D_n containing n labeled pieces of data because the data distribution μ is unknown. Therefore, D_n is usually split into two parts, the *training sample* D_m and the *test sample* D_{n-m}.

A *learning algorithm* is a method that takes the training sample D_m as input and outputs a classifier $f(\mathbf{x}; D_m) = f_m(\mathbf{x})$. A common learning method chooses a function f_m from a function class that minimizes the *training error*

$$L(f, D_m) = \frac{1}{m} \sum_{i=1}^{m} I_{\{f(a_i) \neq y_i\}} \tag{4.1}$$

where I_A is the indicator function of event A. Examples of learning algorithms using this method include the back propagation algorithm for feed-forward neural nets [Rumelhart *et al.*, 1986] or the C4.5 algorithm for decision trees [Quinlan, 1993]. To evaluate the chosen function, the error probability $\Pr_\mu[f(\mathbf{X}) \neq Y]$ is estimated by the *test error* $L(f, D_{n-m})$.

In the following, we provide a short description of the algorithms used to build classifiers in this dissertation: decision trees, naive Bayes classifiers, and logistic regression.

4.6.5.1 Decision Tree

A decision tree is a complete binary tree where each inner node represents a "Yes" or "No" question, each edge is labeled by one of the answers and terminal nodes contain one of the classification labels from the set C.

The decision making process starts at the root of the tree. Given an input vector $\mathbf{x} = (a_1, \ldots, a_d)$, the questions in the internal nodes are answered and the corresponding edges are followed. The label c of $\mathbf{x}$ is determined when a leaf is reached.

In the case of issue classification for example, the leaf node are labeled with either 0 or 1 to indicate whether an issue is a fault or not. The internal node contains question regarding the values of various fields from the issue, for example whether the word "critical" appears in the text describing the issue or if the issue was tagged as "Enhancement".

4.6.5.2 Logistic regression

In a logistic regression model, the dependent variable is commonly a dichotomous variable and, thus, assumes only two values $\{0, 1\}$, *eg.*, changed or not. The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{\beta_0 + \beta_1 \cdot X_1 + \cdots + \beta_n \cdot X_n}}{1 + e^{\beta_0 + \beta_1 \cdot X_1 + \cdots + \beta_n \cdot X_n}} \tag{4.2}$$

where (1) X_j are characteristics describing the modelled phenomenon, for example the number of antipatterns of kind j in a class, *i.e.*, $s_{i,j,k}$ when the model is applied to the class i of release r_k; (2) β_j are the model coefficients; and (3) $0 \leq \pi \leq 1$ is a value on the logistic regression curve. The closer the value is to 1, the higher is the likelihood that the modeled phenomenon occurs.

To use the model as a classifier, a threshold is chosen. For example, in the case of fault classification discussed in Chapters 3 and 8, if the threshold is equal to 0.5, an issue is considered to be a corrective maintenance if $\pi > 0.5$.

4.6.5.3 Naive Bayes Classifier

A Bayesian classifier is a simple classification technique that classifies $\mathbf{x} = (a_1, \ldots, a_d) \in R^d$ by determining its most probable class c computed as:

$$c = \arg max_{c_k} p(c_k|a_1, \ldots, a_d), \tag{4.3}$$

where c_k ranges over the set of classes in C (C $= \{c_1, \ldots, c_q\}$) and the observation $\mathbf{a_i}$ is written as a generic attribute vector. By using *the rule of Bayes*, the probability $p(c_k|a_1, \ldots, a_d)$ called probability *a posteriori*, is rewritten as:

$$\frac{p(a_1, \ldots, a_d|c_k)}{\sum_{h=1}^{q} p(a_1, \ldots, a_d|c_h)p(c_h)} p(c_k). \tag{4.4}$$

The classifier structure is drastically simplified under the assumption that, given a class c_k, all attributes are conditionally independent. Under this assumption the following common form of *a posteriori* probability is obtained:

$$p(c_k|a_1, \ldots, a_d) = \frac{\prod_{j=1}^{d} p(a_j|c_k)}{\sum_{h=1}^{q} \prod_{j=1}^{d} p(a_j|c_h)p(c_h)} p(c_k). \tag{4.5}$$

When the independence assumption is made, the classifier is called naive Bayes classifier. The $p(c_k)$ marginal probability [Fenton and Neil, 1999] is the probability that a member of a class c_k will be observed. The $p(a_j|c_k)$ prior conditional probability is the probability that the j^{th} attribute assumes a particular value a_j given the class c_k. These two prior probabilities determine the structure of the naive Bayes classifier. They are learned, *i.e.*, estimated, on a training set when building the classifier.

A naive Bayes classifier is a simple structure that has (1) the classification node as the root node, to which is associated a distribution of marginal probabilities and (2) the attribute nodes as leaves, to each of them are associated q distribution of prior conditional probabilities, where q is the number of possible classes [Duda and Hart, 1973]. The prior probability is often assumed Gaussian and represented via its mean and standard deviation. In this classifier, discrete and continuous attributes are treated differently [John and Langley, 1995]. In this dissertation, our attributes are discrete. For each discrete attribute, $p(a_j|c_k)$ is a single real that represents the probability that the j^{th} attribute will assume a particular value a_j when the class is c_k.

Continuous attributes however are modeled by some continuous distribution over the range of the attribute. Commonly, it is assumed that within each class, the values of a continuous attribute are distributed as a normal (*i.e.*, Gaussian) distribution represented by its mean and standard deviation. The domain of an attribute j is divided into equally spaced non-overlapping intervals I_{t_j} and an attribute value a_j is interpreted as laying within some interval. Hence, $p(I_{t_j}|c_k)$ represent the prior conditional probability of a value a_j of the j^{th} attribute to be in the interval I_{t_j} when the class is c_k; $t_j \in N$ is the rank of the interval in the attribute domain. This probability is assumed constant on each interval I_{jt_j} and computed based on mean and standard deviation of the Gaussian distribution.

To classify a new observation $a_1, \ldots, a_d$, a naive Bayes classifier with continuous attributes apply Bayes theorem to determine the *a posteriori* probability as follows:

$$p(c_k|I_{t_1}, \ldots, I_{t_d}) = \frac{\prod_{j=1}^{d} p(I_{t_j}|c_k)}{\sum_{h=1}^{q} \prod_{j=1}^{d} p(I_{t_j}|c_h)p(c_h)} p(c_k). \tag{4.6}$$

with $a_j \in I_{t_j}$.

4.7 Summary

In this chapter, after a brief background on design motifs and antipatterns identification techniques, we presented DECOR and DeMIMA; two tools used in this dissertation to identify respectively occurrences of antipatterns and design motifs in systems. We also introduced then POM and Ibdoos frameworks, used respectively to compute metrics and extract commit information from CVS, GIT, or SVN repositories. Finally, we described the 9 open-source object-oriented systems used in our experiments, defined and explained our computation of change and fault-proneness, and discussed the basic concepts underlying the statistical methods used in this dissertation.

Chapter 5

Design Patterns and Quality of Systems

Design patterns are proven solutions to recurrent design problems in object-oriented software design. Their design motifs [Guéhéneuc and Antoniol, 2008] describe *ideal* solutions that will be either used to generate an architecture [Beck and Johnson, 1994] or superimposed [Hannemann and Kiczales, 2002] on designed (or already existing) classes of a system. Consequently, classes in a system may play n roles in m motifs, with $n > 0$, $m > 0$.

Many studies in the literature have for premise that design motifs [Gamma *et al.*, 1994] improve the quality of object-oriented software systems. In addition, it is claimed that every well-structured object oriented design contains patterns, for example [Gamma *et al.*, 1994, page xiii] or [Venners, 2005]. In practice, it is generally observed that although some design patterns ease future enhancements of systems, this ease happens at the expense of simplicity [Venners, 2005]. Evidence of quality improvements through the use of design patterns consists primarily of intuitive statements and examples. Moreover, some studies, *eg.*, by Wendorff [2001], have suggested that the use of design patterns, *i.e.*, composition of motifs, does not always result in "good" designs. McNatt and Bieman [2001], hinted that tangled implementation of patterns may rather in fact decrease the quality of systems. Hannemann and Kigzales [2002] pointed out the difficulty to reason about classes involved in several design motifs. They claimed that from the 23 design motifs presented in the catalog [Gamma *et al.*, 1994], 17 of them lack modularity and are invasive in systems. In the first pilot study, presented in Chapter 3, we reported that design patterns are sometimes perceived as negatively impacting the quality of systems.

All these studies suggest that design motifs impact the quality of systems; different motifs impacting the quality differently. Therefore to effectively assess the quality of a system, we should not only consider the classes of the system independently but also their organisation in micro-architectures similar to motifs. Following our method DEQUALITE presented in Chapter 1, we must quantify the impact of design motifs on the quality of systems. We present in the following the results of an empirical study aimed at assessing the impact of design patterns on the change- and fault-proneness of classes in a system. The quantitative information obtain from this study will be used in the quality models presented in Chapter 8.

5.1 Context

Previous work considered that classes either play *no* role or *some* role(s) in *some* motif(s), without distinguishing classes playing one or more roles in one or more motifs. For example, Bieman *et al.* [2001b] and Di Penta *et al.* [2008] suggested that classes playing one specific role in one design motif may be more complex and more change-prone than classes playing no roles *because* of their role but without considering the possibility that a class could play two or more roles. These previous work neglected that classes may play many different roles and took the perspective that role playing is an *all-or-nothing* characteristic of classes.

This coarse-grained perspective prevents studying *finely* the impact of role playing on classes. A reason for the current coarse-grained perspective is the lack of a method and manually-validated data to identify and evaluate the characteristics of classes playing one, two, or more roles with respect to classes playing no role. Yet, studying finely number of roles could provide supporting evidence to previous work concerned by the use and abuse of design patterns and would allow improving quality models with this new information.

Therefore, in this chapter, we describe a method and present an empirical descriptive and analytic study of the impact on classes of playing one role in a motif or two roles in a motif composition. The method is essential to obtain statistically significant results. The descriptive study shows that, indeed, a non-negligible proportions of classes play one or two roles in systems and that some roles are more often played in pairs than others. The analytic study shows that some internal and external characteristics of classes are impacted by playing one or two roles and that playing two roles impact more classes than playing one role. Thus, we bring quantitative evidence on the impact of design motifs in systems, which has only been hinted at until now. This evidence allows us to make some

observations on motif composition and to draw some practical conclusions on the use and abuse of design patterns. We also revisit previous work on design patterns with this novel fine-grain perspective.

5.2 Study Definition and Design

Following GQM [Basili and Weiss, 1984], the *goal* of this chapter is to present a study of classes playing zero, one, or two roles in some design motifs. Our *purpose* is to bring generalisable, quantitative evidence on the impact of playing roles on classes and build quality models. The *quality focus* is that playing zero, one, or two roles impact differently classes. The *perspective* is that both researchers and practitioners should be aware of the impact of playing roles on classes to make informed design and implementation choices. The perspective is also to build quality models, and to understand and forecast the characteristics of classes. The *context* of our study is both development and maintenance.

5.2.1 Research Questions and Hypotheses

Descriptive Questions. The two first research questions are descriptive and aim at understanding the extent to which classes play zero, one, or two roles in a general population of classes. If the proportions are not negligible, then the two following analytic questions will be meaningful. These research questions are particularly important because large proportions of one and two roles would mean an extensive use of design patterns in systems and therefore a potential high impact on the quality of the system.

- **RQ1:** *Given a population of classes, what is the proportion of classes playing zero, one, or two roles in some motif(s)?*
- **RQ2:** *What are the roles that are more often played solitary or in pairs than others?*

Analytic Questions. The two following questions are analytic and divide in two sets of null hypotheses.

- **RQ3:** *What are the internal characteristics of a class that are the most impacted by playing one or two roles w.r.t. playing less roles?*
- **RQ4:** *What are the external characteristics of a class that are the most impacted by playing one or two roles w.r.t. playing less roles?*

For any metric m measuring some internal or external characteristics of a class, we test the set of null hypotheses: $H_{0mi/j}$*: the distribution of the values of metric* m *for the classes playing* $i \in [1,2]$ *role(s) is similar to that of classes playing* $j \in [0,1] \wedge i \neq j$ *role.* We relate the following independent and dependent variables to assess the proportions of classes playing different roles and to test the previous null hypotheses.

5.2.2 Independent Variables

In an ideal situation, we would know the general population of *all possible classes* and know the number of roles played by any class. Then, we would use the sub-populations of classes playing zero, one, or more roles to answer the research questions. However, this situation is impossible because the population of all possible classes is infinite and, in general, a class does not know if it plays any role.

Therefore, the independent variables are three samples of classes playing zero, one, and two roles in design motifs. We limit our study to two roles to ease the manual validation. We name these samples the 0-, 1-, and 2-role samples. The samples must be large enough to be statistically representative but small enough to be manually inspected. The method to build these samples along with its implementation are presented in Section 5.3.

5.2.3 Dependent Variables

The dependent variables are the metrics measuring classes internal and external characteristics. We chose to study a large number of metrics, as previous work [Spinellis, 2008], to assess all the possible impacts of role playing. **Internal Characteristics** are related to class themselves and are measured using the 56 metrics presented in Section 4.3. **External Characteristics** are limited in this study to the change- and fault- proneness of classes defined in Section 4.4.

5.2.4 Descriptive and Analytic Analyses

We use the following analysis to answer the research question with independent and dependent variables.

RQ1 and RQ2. Given a population of classes from 6 systems, we computed the classes playing zero, one, and two roles with our identification approach DeMIMA presented in Chapter 4. Then, we computed the accuracy of our approach for one and two roles by

manually validating classes playing roles in the identified occurrences. This precision is important to estimate the proportions of true positives one and two roles classes. With this precision, we extrapolate the proportions of classes playing zero, one, and two roles in the general population.

RQ3 and RQ4. We use the Wilcoxon rank-sum test to compute for each metric and each pair of samples (0-role, 1-role), (0-role, 2-roles), and (1-role, 2-role), the p-values for the corresponding null hypotheses. As discussed in Chapter 4, this test allows us to reject the null hypotheses while making no assumptions on the normality of the samples.

5.3 Study Implementation

The following subsections detail the building of the samples.

5.3.1 Definitions

We define a:

- **General population** as the set of all classes and interfaces belonging to some given systems;
- **n-role population** as the population of classes playing n roles in some design motifs. Thus, the *0-role* population contains all the classes in the general population playing no role. The *1-role* population contains classes playing one and only one role. The *2-role* population includes only classes playing two roles in two different motifs, *i.e.*, playing roles in *pairs of motifs*;
- **n-role class subset** as a subset of the classes in the general population that has been manually studied to identify n-role classes;
- **n-role sample** as the intersection of the n-role class subset and the n-role population: a manually validated sample of n-role classes.

Figure 5.1 illustrates the partition of a general population of classes. We define three sub-populations, which form a partition of the general population. The *0-role* population contains all classes playing no role. The *1-role* population contains classes playing one and only one role. The *2-role* population includes only classes playing two roles in two

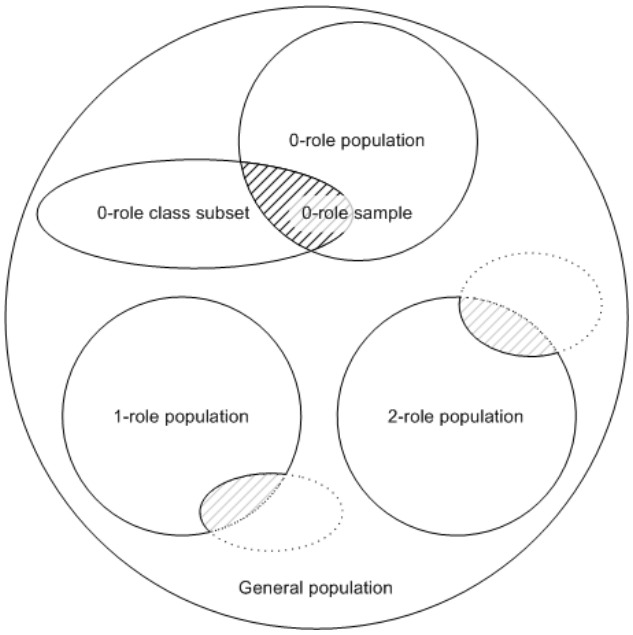


Figure 5.1 – Subsets of the general population, details are given for 0-role classes.

different motifs, *i.e.*, playing roles in *pairs of motifs*. We extracted from the 0-, 1-, and 2-role populations three subsets of classes, CS_0, CS_1, and CS_2, that we manually validated to build, after validation, the 0-, 1-, and 2-role samples with which we will answer the research questions. We use different n-role class subsets when identifying classes playing n role(s) to avoid any bias.

5.3.2 Sizes of the Samples

The size of the samples must be large enough to allow the generalisation of the results to the overall population yet small enough to be validated manually.

We compute the sample size in two steps as proposed by Hollander and Wolfe [1999]:

1. We first compute the sample size needed for a two-sample t-test; and next,
2. We adjust this size based on the Asymptotic Relative Efficiency (ARE) [Hollander and Wolfe, 1999] of the two-sample Wilcoxon test.

We choose a *typical power of* 0.8, *i.e.*, we seek 80% chance of finding statistical significance if the specified effect exists. The statistical power tells us, in probability terms,

the capability of our test to detect a significant effect. It tells us how often we can reach a correct interpretation about the effect, if we would be able to repeat the test many times.

We also choose a *typical significance level of* 0.05 because we seek to reduce the possibility that the probability is due to chance alone.

With this power and significance level, we study the relation between effect size and sample size to choose the adequate sample size for a two-sample t-test, assuming the normality of the distribution.

The effect size refers to the magnitude of the effect under the alternate hypothesis. The choice of an effect size reflects the need for balance between the size of the effect that we can detect and the resources available for the study. We plotted the values of the samples sizes corresponding to the effect size varying from 0.5 to 1.5. Figure 5.2 presents the obtained curve. We can observe that small effects require a larger investment in resources (larger sample sizes) than large effects.

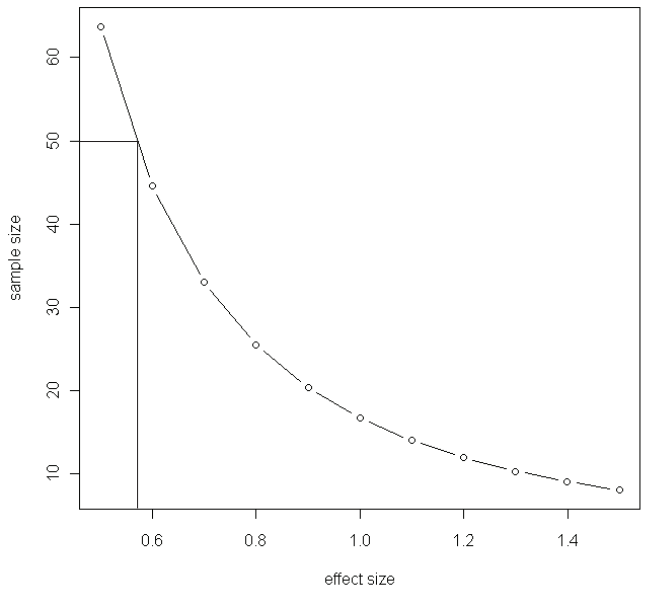


Figure 5.2 – Possible sample sizes *w.r.t.* effect size for the t-test.

To decide on effect size, we followed Cohen's recommendations [Cohen, 1988] and chose a *medium effect size of 0.58* that corresponds to a sample size of 50 classes.

The ARE represents the asymptotic limit of the ratio of the samples sizes needed to achieve equal power for two statistical tests: given a sample size for a statistical test A achieving a power p, the sample size needed for a test B to achieve the same power p is obtained from the ARE of A *w.r.t.* B. We compute the sample size for the two-sample Wilcoxon test that ensures the same power as the t-test, with no assumption of the distribution. The ARE for the two-sample Wilcoxon test is never less than 0.864 [Hollander and Wolfe, 1999], we choose to be conservative and therefore divide the sample size for a t-test by 0.864. We obtain a *sample size of 58 classes*.

Consequently, the parameters of our study are:

- **Power**: 0.8 (typical);
- **Significance level**: 0.05 (typical);
- **Effect size**: 0.58 (medium);
- **Sizes of the samples**: 58 classes.

5.3.3 Selection of the General Population

We choose six systems to form the general population of classes from which to build the n-role samples: ArgoUML v0.18.1, Azureus v2.1.0.0, Eclipse JDT Core plug-in v2.1.2 (JDT Core v2.1.2), JHotDraw v5.4b2, Xalan v2.7.0, and Xerces v1.4.4. These systems are written in Java and open source. They are of different domains, sizes, complexity, maturity and have been used in previous studies [Aversano *et al.*, 2007 ; Di Penta *et al.*, 2008 ; Olbrich *et al.*, 2009]. Section 4.5 presents their descriptions and Table 5.1(a) summarises facts on these systems.

5.3.4 Selection of the Motifs and their Roles

We selected six design motifs used in previous work by Tsantalis *et al.* [2006] and Di Penta *et al.* [2008]: Command, Composite, Decorator, Observer, Singleton, and State. We follow Di Penta *et al.* [2008] in their choice of the motifs *main* roles. We only study main roles because (1) they are the ones implementing the core functionalities of motifs and, therefore, are most likely to impact classes, as confirmed by the following results and (2) they allow us to concentrate on a fewer number of roles during the manual validation. In the following, roles are named using the notation <Pattern Name> . <Role Name>.

Table 5.1 – Data on the Studied systems.

(a) Statistics for the six systems. (Future refers to the time between the release dates and 31/01/09.)

systems	Classes	LOC	Release Dates	Past Changes	Future Changes	Issues
ArgoUML v0.18.1	1,267	202,520	30/04/05	20,290	12,617	41,565
Azureus v2.1.0.0	591	83,534	1/06/04	18,304	483	33,753
JDT Core v2.1.2	669	184,690	3/11/03	23,243	26,923	62,728
JHotDraw v5.4b2	413	44,898	1/02/04	5,793	51	1,286
Xalan v2.7.0	734	259,286	8/08/05	12,298	1,714	58,448
Xerces v1.4.4	306	86,814	13/10/03	5,213	1,209	16,143
Total	3,980	861,742	6 releases	85,141	42,997	213,923

(b) Distribution of the sample size among the systems of our strata.

systems	Expected	Zero Role	One Role	Two Roles
ArgoUML v0.18.1	17	17	10	10
Azureus v2.1.0.0	9	9	9	9
JDT Core v2.1.2	10	10	17	17
JHotDraw v5.4b2	6	6	6	6
Xalan v2.7.0	11	11	11	11
Xerces v1.4.4	5	5	5	5
Total	58	58	58	58

Table 5.2 – Chosen design patterns and the main roles of their motifs.

Patterns	Descriptions	Main Roles
Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations	Command, Invoker
Composite	Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly	Component, Composite
Decorator	Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality	Component, Decorator
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically	Observer, Subject
Singleton	Defines a mechanism that ensure that the same instance of a class is used throughout a system execution	Singleton
State	Allows an object to alter its behavior when its internal state changes	Context, State

In addition to choosing the roles of interest, we must also select pairs of roles for classes playing two roles. We excluded pairs with the same role because identical roles in different motifs must have similar characteristics as they implement the same functions, *eg.*, among the six motifs, Component is the only role that appears twice with similar structure albeit slightly different semantics. We excluded pairs involving roles from the same motif because a class playing both the roles of Composite.Component and Composite.Composite must be a degenerated case. Consequently, we retain 45 possible pairs.

5.3.5 Building of the Samples

Building the n-role sample, with $n \in [0, 2]$, consists of searching in the general population for three sets of 58 classes playing 0, 1, or 2 roles. We reduce the search space using DeMIMA approach because it ensures 100% recall by automatically relaxing appropriate constraints. DeMIMA provides a complete mapping between roles in a motif and classes in an occurrence. Thus, we can find all the roles that each class plays for a set of motifs in a system.

We applied DeMIMA on the classes in the general population and obtain candidate classes playing (at least) one role in the selected motifs. We automatically divided this set in two 1- and 2-role subsets.

Then, for each subset, we studied each class (its code source, comments, hierarchy, relations) to decide whether it plays one role (respectively two roles) using a voting process:

four developers marked independently each class as *true* when a class played one role (respectively, two roles) or $false$ else. Each class was marked by only three developers to avoid ties. Then, a class was assigned to the 1-role sample (respectively, 2-role sample) if the majority marked it as *true*, else it was excluded. We stopped the voting process as soon as the samples were completed.

In total, 238 classes were manually validated: 81 classes where false positives, *i.e.*, classes playing no role but belonging to occurrences identified by DeMIMA; 88 classes played 1 role; and, 69 classes played 2 roles. Finally, from the classes *not* included in any of the occurrences identified by DeMIMA, we selected randomly and validated manually 58 classes playing 0 role.

The distribution in the samples of the classes from the general population must be representative of the population. We distributed the 58 classes per sample along the strata formed by the six systems. We computed stratified sample sizes so that each stratum reflected the proportional size of one system with respect to the others. For example, JHotDraw v5.4b2 makes up 10.38% of the general population. So, it must provide 10.38% of the 58 classes in each sample. Thus, we ensured that the results equally reflect the six systems. The second column in Table 5.1(b) shows the expected size of each stratum, *i.e.*, the expected numbers of classes of each system in each sample.

We could not find enough 1- and 2-role classes in ArgoUML. Therefore, we made up for the reduced number of occurrences of motifs in ArgoUML by using more classes from JDT Core. The fourth and fifth columns in Table 5.1(b) show the actual repartitions of classes in the 1- and 2-role samples. We replicated our study on the general population without JDT Core and on JDT Core exclusively and noticed the same trends.

5.4 Study Results

We analyse the metrics values computed on the classes in the samples to answer the research questions.

5.4.1 RQ1: Given a population of classes, what is the proportion of classes playing zero, one, or two roles in some motif(s)?

To answer our first research question "What is the proportion of classes playing zero, one, or two roles in some motifs in a system?", we extrapolate, for each system and each motif, the number of classes playing zero, one role, and two roles in the motifs.

Table 5.3 – Validated precisions of DeMIMA.

systems	Candidates	One Role	Two Roles
ArgoUML v0.18.1	21	3	10
		14.28%	47.61%
Azureus v2.1.0.0	64	22	19
		34.37%	29.68%
JDT Core v2.1.2	67	30	22
		44.77%	32.83%
JHotDraw v5.4b2	30	11	13
		36.66%	43.33%
Xalan v2.7.0	55	23	11
		41.81%	20.00%
Xerces v1.4.4	29	10	11
		34.48%	37.93%
Total	266	99	86
		37.21%	32.33%

Table 5.4 – Extrapolated numbers and percentages of classes playing no, one, or two roles.

systems	Total	One Role	Two Roles
ArgoUML v0.18.1	1,267	51	316
	100%	4.02%	24.94%
Azureus v2.1.0.0	591	67	75
	100%	11.33%	12.69%
JDT Core v2.1.2	669	46	178
	100%	6.88%	26.60%
JHotDraw v5.4b2	413	24	101
	100%	5.81%	24.45%
Xalan v2.7.0	734	36	104
	100%	4.90%	14.16%
Xerces v1.4.4	306	94	56
	100%	30.72%	18.30%
Total	3,980	318	830
	100%	7.99%	20.85%

First, from the class subsets, we computed the accuracy of DeMIMA as the number of classes in a subset *indeed* playing zero, one, or two roles with respect to the total numbers of classes in the subsets. This accuracy is important to estimate the proportion of classes playing one and two roles respectively. Table 5.3 summarises this accuracy using all manually-validated classes (reported in the Candidates column) and shows that accuracy varies across motifs and systems.

Second, we extrapolate in Table 5.4 the numbers of classes playing one and two roles from the previous accuracy and the numbers of candidate classes in each system. Table 5.4 shows that the percentage of classes playing one or two roles in any of the six selected design motif varies from 4.02% to 30.72%.

The answer to RQ1 is that classes playing one or two roles do *exist in systems and are not negligible, which confirms the need to understand the characteristics of classes playing different numbers of roles as they may significantly impact the quality of systems.*

5.4.2 RQ2: What are the roles that are more often played solitary or in pairs than others?

An answer to the research question "What are the roles that are more often played solitary or in pairs than others?" is obtained by studying the proportions of the numbers of classes playing one or two roles with respect to the number of classes playing a particular role.

Table 5.5 shows the numbers and proportions for each role and each pair for which the proportions of classes playing this role or pair of roles was not negligible. It shows that, for example, five classes play the role of Command.Command alone while four play the roles of both Command.Command and State.State, which make up for 44.44% of classes playing both roles with respect to the $5 + 4$ classes playing the Command.Command role. This information about roles often played solitary or in pairs is interesting because it can help guide and improve developers comprehension of systems as pointed by Lange and Nakamura [1995]. Moreover, this information can help improve the precision and recall of design motifs identification tools: A tool that detected a motif A, which frequently appeared with motif B, will be able to improve his detection of Motif B by considering the frequency of the pair (A, B).

We notice three facts:

1. There are three pairs and two solitary roles for which the percentage is above our decision threshold: (Command.Invoker, State.State), (Decorator.Component, State.-State), (Decorator.Decorator, State.State), Composite.Composite, State.Context.

Table 5.5 – Counts and percentages of roles played alone or paired with another role.

Roles	Pairs	Counts	Percentages
Command.Command		5	55.56%
	(Command.Command , State.State)	4	44.44%
Command.Invoker		0	0%
	(Command.Invoker , State.State)	13	**100%**
Composite.Component		4	22.22%
	(Composite.Component, Observer.Observer)	9	50%
	(Composite.Component, State.State)	5	27.78%
Composite.Composite		8	**80%**
	(Composite.Composite , State.Context)	2	20%
Decorator.Component		1	11.11%
	(Decorator.Component , State.State)	8	**88.89%**
	(Decorator.Component , State.Context)	0	0%
Decorator.Decorator		1	8.33%
	(Decorator.Decorator , State.State)	11	**91.67%**
Observer.Observer		36	66.67%
	(Composite.Component, Observer.Observer)	9	16.67%
	(Observer.Observer , State.State)	9	16.67%
State.Context		33	**94.29%**
	(Composite.Composite , State.Context)	2	5.71%
	(Decorator.Component , State.Context)	0	0%
State.State		**50**	**50%**
	(Command.Command , State.State)	4	4%
	(Command.Invoker , State.State)	13	13%
	(Composite.Component, State.State)	5	5%
	(Decorator.Decorator , State.State)	11	11%
	(Decorator.Component , State.State)	8	8%
	(Observer.Observer , State.State)	9	9%

Table 5.6 – Selected pairs of roles.

Pairs
(Command.Command , State.State)
(Command.Invoker , State.State)
(Command.Invoker , Singleton.Singleton)
(Composite.Component, Observer.Observer)
(Composite.Component, State.State)
(Composite.Component, Singleton.Singleton)
(Composite.Composite , State.Context)
(Composite.Composite , Singleton.Singleton)
(Decorator.Component , State.Context)
(Decorator.Component , State.State)
(Decorator.Component , Singleton.Singleton)
(Decorator.Decorator , State.State)
(Observer.Observer , State.State)
(Singleton.Singleton , State.State)

We could conclude that these pairs are prevalent. Yet, these roles and pairs are only played by a few classes.

2. Among the roles/pairs with a significant number of classes (more than 50) playing these roles, the percentages are smaller than our decision threshold and prevents us to generalise our results.

3. The preponderance of pairs involving roles in the State motif possibly indicates a bias during the manual validation of the classes. We further discuss the threat to the validity of our study in Section 5.6.

We therefore answer that, in the six studied systems, pairs (Command.Invoker, State.-State), (Decorator.Component, State.State), (Decorator.Decorator, State.State) and roles Composite.Composite and State.Context have greater prevalence than others, which confirms that some roles are more often played together than others.

5.4.3 RQ3: What are the internal characteristics of a class that are the most impacted by playing one or two roles *w.r.t.* playing less roles?

We now limit our selection of roles to the 14 pairs shown in Table 5.6 by keeping only pairs for which we had enough classes, determined in RQ1.

To answer RQ3, "What are the internal characteristics of a class that are the most impacted by playing one or two roles?", we test the null hypotheses $H_{0mi/j}, i \in [1,2], j \in [0,1] \wedge j \neq i$ for the 56 metrics.

Table 5.7 summarises the results. It shows for each metric in each metric group the p-value when testing the associated null hypothesis. It reports in bold the p-values that show a statistically significant difference between the distribution of the metric values between two samples. It also shows using arrows the trend in the change between samples. Although we performed a Holm-Bonferroni correction on the p-values and obtained little variations on the results, we choose to report the original results in Table 5.7 because the Holm-Bonferroni correction increases the risk of Type II errors.

We analyse the results in Table 5.7 in three steps: metrics whose distributions do not change between samples and then those that change between each pairs of samples:

5.4.3.1 Metrics Not Affected by The Number of Roles

There are 8 metrics whose distributions did not change significantly between the three samples: connectivity, CP, PP, RPII, DSC, ANA, NOH, and MFA. These metrics are therefore unlikely to be of interest when assessing the impact of role playing and could be excluded from future studies on design motifs.

This finding was predictable for CP, PP, RPII because these metrics measure the structure of the packages of a system rather than the structure of its classes. The same explanation applies to DSC and NOH, which count respectively the total number of classes and the number of class hierarchies in a system.

The finding for ANA, connectivity, and MFA is surprising because we expected that classes playing roles in design motifs would inherit more from and would be more "connected" to other classes. We explain this finding by the specific definitions of these three metrics because the values of other metrics related to inheritance and coupling significantly change between the samples.

5.4.3.2 Metrics Affected by One Role vs. Zero Role

There is a statistically significant difference between classes playing zero and one role for 29 metrics. These metrics characterise coupling, cohesion, inheritance, polymorphism and size, and complexity.

Table 5.7 – p-values and Metrics Trends. (A ↗ or ↘ represents an increase (respectively, decrease) of, for example in the third column, the metrics values of 1-role classes compared to these of 0-role classes).

Metric Groups	Metric Names	1 role vs. 0 role		2 role vs. 0 role		2 role vs. 1 role	
		p-values	Trends	p-values	Trends	p-values	Trends
Changeability	Frequencies of Past Changes	**8.26E-07**	↗	**1.24E-09**	↗	0.08794	
	Frequencies of Future Changes	**0.0001564**	↗	**7.44E-06**	↗	0.5983	
	Numbers of Past Changes	**3.54E-07**	↗	**5.50E-10**	↗	0.06668	
	Numbers of Future Changes	**0.001552**	↗	**9.72E-05**	↗	0.7018	
Cohesion	CAM	0.854		**0.0001996**	↗	**0.0003884**	↗
	cohesionAttributes	0.6881		**0.04051**	↗	**0.0009488**	↗
	LCOM1	**0.01313**	↘	**6.22E-09**	↗	**0.0009946**	↗
	LCOM2	**0.01087**	↘	**1.41E-07**	↗	**0.0017**	↗
	LCOM5	**0.03454**	↗	**3.95E-06**	↗	**0.001383**	↗
Complexity	McCabe	0.2274		**7.85E-07**	↗	**0.00063**	↗
	SIX	**0.004657**	↗	**1.41E-08**	↗	**0.0008183**	↗
	WMC1	**2.09E-05**	↗	**4.00E-08**	↗	**0.0467**	↗
	WMC	**0.01453**	↘	**5.40E-07**	↗	**0.001297**	↗
Coupling	ACAIC	0.1733		**0.03935**	↗	0.5029	
	ACMIC	0.284		**0.002702**	↗	**0.04961**	↗
	CBO	0.5706		**0.0001434**	↗	**0.001948**	↗
	CBOin	0.191		**7.89E-06**	↗	**0.0005939**	↗
	CBOout	0.1055		**5.96E-07**	↗	**0.0001025**	↗
	connectivity	0.5005		0.07963		0.2603	
	CP	0.9802		0.2272		0.1428	
	DCAEC	**9.37E-06**	↗	**0.003612**	↗	0.06724	
	DCC	0.4149		**2.98E-05**	↗	**0.002347**	↗
	DCMEC	**0.0001468**	↗	**0.001024**	↗	0.595	
	PP	0.829		0.1382		0.1468	
	RFP	**0.04845**	↗	**0.01477**	↗	0.6074	
	RRFP	0.0968		**0.02306**	↘	0.5106	
	RRTP	**0.02637**	↘	**0.03722**	↘	0.6952	
	RTP	0.2005		**0.01295**	↗	0.3693	
Inheritence	AID	0.126		**0.0001542**	↗	0.1391	
	ANA	0.3958		0.8077		0.3918	
	CLD	**< 2.2e-16**	↗	**7.94E-11**	↗	**0.003298**	↘
	DIT	0.08713		**8.59E-05**	↗	0.2632	
	NCM	**0.00087**	↗	**4.84E-09**	↗	0.07486	
	NOC	**2.22E-16**	↗	**3.55E-11**	↗	0.245	
	NOD	**2.22E-16**	↗	**5.29E-11**	↗	0.07351	
	NOH	0.5644		0.601		0.9663	
	NOP	0.2248		**6.10E-06**	↗	**0.007146**	↗
	ICHClass	**0.03035**	↗	**2.03E-07**	↗	**0.001095**	↗
Issues	Numbers of Issues	**0.0003619**	↗	**0.0003612**	↗	0.6645	
Polymorphism and Size	CIS	**9.22E-07**	↗	**1.50E-08**	↗	0.1605	
	DAM	0.1285		**1.94E-05**	↗	**0.003362**	↗
	DSC	0.1461		0.2098		0.8725	
	EIC	**0.0002848**	↗	**9.03E-06**	↗	0.5616	
	EIP	**7.26E-13**	↗	**1.43E-09**	↗	0.1039	
	MFA	0.1138		0.7105		0.243	
	MOA	**0.0001883**	↗	**6.44E-10**	↗	**0.01493**	↗
	NAD	0.1349		**5.03E-06**	↗	**0.003884**	↗
	NADExtended	0.1514		**1.14E-05**	↗	**0.005466**	↗
	NCP	**5.39E-06**	↗	**0.01465**	↗	0.1198	
	NMA	**9.34E-06**	↗	**2.30E-06**	↗	0.3157	
	NMD	**2.09E-05**	↗	**4.00E-08**	↗	**0.0467**	↗
	NMDExtended	**3.37E-05**	↗	**1.07E-07**	↗	0.05112	
	NMI	0.1029		**0.0001075**	↗	0.2016	
	NMO	**0.00163**	↗	**3.57E-10**	↗	**0.0005408**	↗
	NOA	0.1868		**7.35E-08**	↗	**0.01153**	↗
	NOM	**2.09E-05**	↗	**4.00E-08**	↗	**0.0467**	↗
	NOParam	**7.81E-06**	↗	**2.38E-08**	↗	0.1551	
	NOPM	**2.89E-14**	↗	**1.93E-10**	↗	0.2793	
	PIIR	**7.00E-05**	↗	**0.01216**	↗	0.2846	
	REIP	**5.94E-10**	↗	**7.54E-08**	↗	0.3336	
	RPII	0.1486		0.08605		0.8614	
Ranking	Class Rank	**7.33E-09**	↗	**4.08E-06**	↗	0.212	

The trends are a decrease in metric values for only four metrics: LCOM1, LCOM2, WMC1, and RRTP. This finding is explained again by the implementations of the metrics: LCOM1 and 2 have been superseded by LCOM5, which changes significantly, while WMC1 weighs each method by 1 and RRTP is related to packages.

The others metrics see a statistically significant increase in their values. Among these, we can quote: CBO, DCAEC, LCOM5, McCabe, SIX, WMC. We explain this finding by the fact that playing roles implies responsibilities, thus classes playing one role have more responsibilities than classes playing zero role, which results in classes being more complex (McCabe, SIX, WMC), more coupled (CBO, DCAEC), and less cohesive (LCOM5), for example. We conclude that playing one role impact classes *w.r.t.* playing zero role.

5.4.3.3 Metrics Affected by Two Roles vs. Zero Role

There is a statistically significant difference between classes playing zero and two roles for 48 metrics, with, for each metric, an increase of its values for classes playing two roles, except for RRFP and RRTP. This finding was expected because RRFP and RRTP concern packages. For the 46 other metrics, the added responsibilities with each role could explain the impact of 2-role classes on metric values in comparison to the impact of classes playing zero role. Having more responsibilities, classes become more complex (McCabe, WMC, WMC1, SIX), more coupled (CBO, DCAEC, DCC, DCMEC), inherit more from their superclasses (CLD, DIT, NOC, NOD), and use more polymorphism (MOA, NMA, NMD). Therefore, we conclude that playing two roles has a major impact on classes, in particular in comparison to the impact of playing zero role. Playing two roles should be carefully considered during design and implementation.

5.4.3.4 Metrics Affected by Two Roles vs. One Role

The change in the distributions of the metrics values between classes in the 2- and 1-role samples is significant for 26 metrics, among which: CAM, CLD, DCC, LCOM5, McCabe, SIX, WMC. We observe that the more they play roles, the more classes are complex (McCabe, SIX, WMC, WMC1), are coupled (CBO, DCC), inherit (NOP), and use polymorphism (MOA, NAD, NMO).

The values of CLD decrease significantly, possibly hinting at more shallow inheritance tree thanks to the solutions provided by the motifs. We conclude that, indeed, playing two roles has a significant impact on classes that cannot be accounted for by the fact that they play two different one roles.

Consequently, the answer to RQ3 is that, w.r.t. the studied metrics, playing two roles has a major impact on classes when compared to playing zero or one role.

5.4.4 RQ4: What are the external characteristics of a class that are the most impacted by playing one or two roles *w.r.t.* playing less roles?

We answer the last research question, "What are the external characteristics of a class that are the most impacted by playing one or two roles?", by carrying null hypothesis tests on the numbers and frequencies of past and future changes, extracted from the version repositories of the systems, and on the numbers of issues related to classes in the different samples. Table 5.7 shows the results of testing out the null hypotheses.

We can reject the null hypotheses related to the external metrics (Changeability and Issues) for 1-role and 2-role classes *w.r.t.* 0-role classes with statistical significance. We cannot reject the null hypotheses for 2-role classes when compared to 1-role classes.

These results confirm previous work on the change- and issue-proneness of classes playing roles in some design motifs, for example [Bieman *et al.*, 2001b ; Di Penta *et al.*, 2008]. We perform in Section 6.4 a deeper analysis that shows that 2-role classes are the cause of the greater parts of the changes (56%) and issues (57%) with 1-role classes causing only 33% of changes and 30% of issues.

The answer to RQ4 is that playing roles do impact the number of changes and issues as well as the frequencies of the changes. It confirms that playing roles has a major impact on change- and issue-proneness and therefore on the quality of systems. In Chapter 8, we include this information in our quality models.

5.5 Discussions

With the results of our study, we revisit previous work from the literature presented in Chapter 2; first, to validate our results and, second, to show how these previous work could benefit from our fine-grained analysis.

5.5.1 Proportions of 0-, 1-, or 2-role Classes

Table 5.4 shows the percentages of classes playing no, one, or two roles in the six systems. In addition to the overall percentages, some systems have higher percentages than others:

JHoDraw contains *only* 5.81% of classes playing one role and 24.45% two roles in contrast to the 30.72% of classes playing one role in Xerces and the 26.60% of classes playing two roles in JDT Core. Given that JHotDraw has been developed to show the "good" use of design patterns, the higher percentages of classes playing one or two roles in Xerces and JDT Core could be due to an *overuse* of design patterns. These higher percentages could be used with other quality measures to confirm or refute the impact of overusing design patterns as put forward by Wendorff [2001] and others. In Chapter 8, we show that this information on design patterns improves the accuracy of quality models.

5.5.2 Trends in Playing Roles and Quality

Table 5.7 shows that in the 2-role sample, classes are more complex, more coupled, less cohesive than classes in the 0- and 1-role samples. This trend suggests that motif composition (playing more than one role in some motifs) degrades more the quality of the classes than playing a single role. We explain this degradation by the addition to the classes of non-feature methods and fields to allow the classes to fulfill their roles. These methods and fields increase the complexity, the number of dependencies, and reduce the cohesion of classes.

5.5.3 Revisit of and Comparison with Previous Work

5.5.3.1 Bieman and McNatt's Work

We observe that playing one or more roles in a design motif decreases the cohesion of classes (increases of the LCOM$\star$ metrics) while increasing their coupling (increase of the coupling metrics). This result confirm Bieman and McNatt's [2001] claim that design motifs impact the cohesion and coupling of systems.

5.5.3.2 Hannemann and Kicazles' Work

We explain the decrease in cohesion and increase in coupling by suggesting that design motif-related methods may be orthogonal to the responsibilities of the classes and thus reduce their cohesion. Therefore, our study confirms that design motifs are often "cross-cutting concern" that could benefit from being "separated" from the system using, for example, aspect-oriented programming. We thus bring quantitative support to previous work on rewriting design motifs as aspects [Hannemann and Kiczales, 2002].

5.5.3.3 Di Penta *et al.*'s Work

We revisit Di Penta *et al.*'s study of the numbers and frequencies of changes of classes playing roles. We compare the set of classes playing some roles, as identified by DeMIMA, which is the union of the samples of 1- and 2-role classes with the sample of false positive classes, noted 0^{FP}, with the set of classes playing *really* zero role: 0-role sample vs. (0^{FP}-role $\cup$ 1-role $\cup$ 2-role) sample. This comparison yields a p-value of **1.973e-14** < 0.05, thus confirming the previous work as well as the statistical validity of our three samples.

It appears from our study that, in average, the numbers of changes prior to the releases of the studied system for classes playing two roles accounts for 56% of the total number of past changes. Also, classes playing two roles change 1.52 times more than classes playing one role. Classes playing zero and one role account respectively for 33% and 11% of past changes. Classes playing one role change more than two role classes after the studied release of the systems. They change 1.46 times more than the 2-role classes and they account for 61.53% of the total number of future changes. We explain this result by the fewer numbers of future changes, shown in Table 5.1(a): in total, there are twice as much past changes than future changes. Therefore, we bring evidence that the results found by Di Penta *et al.* was largely due to classes playing two roles.

5.5.3.4 Automatic Detection with DeMIMA

Table 5.8 shows the fine-grained study of the impact for a class to be a false positive *w.r.t.* playing zero, one, or two roles. It shows that false positive classes do have a significantly different number of changes than classes playing 0 role. This results was expected because false positive classes must have some particular feature: DeMIMA included them in its results. A typical case is the Composite: A motif with a structure close to the Composite motif but with an inheritance instead of a composition may be detected by DeMIMA. In this case, we expect that playing a role in this false Composite would affect a class. Results in Table 5.8 show that classes playing two roles change significantly differently from false positives classes, thus confirming their importance. Therefore, although these classes play no role in a design motif, their structure is somehow similar to some design motifs, which impacts their internal and external characteristics. Thus, the presence of false positives classes in systems impacts the system overall quality. This result is particularly interesting because it shows that automatic detection tools like DeMIMA can help provide useful information which we will use in Chapter 8 to include in a novel quality model.

Table 5.8 – p-values and Metrics Trends, with 0^{FP}. (A ↗ or ↘ represents an increase (respectively, decrease) of, for example in the third column, the metrics values of 1-role classes *w.r.t.* to these of 0-role classes).

Metric Groups	Metric Names	1 role vs. 0 role		2 role vs. 0 role		2 role vs. 1 role	
		p-values	Trends	p-values	Trends	p-values	Trends
Changeability	Frequencies of Changes in Past	0.9956		0.08194		0.08794	
	Frequencies of Changes in Future	0.9733		0.5469		0.5983	
	Numbers of Changes Past	0.212		**0.03508**	↗	0.06668	
	Numbers of Changes Future	0.8688		0.8537		0.7018	
Cohesion	CAM	0.8532		**0.0007399**	↗	**0.0003884**	↗
	CohesionAttributes	0.5716		**0.01112**	↗	**0.0009488**	↗
	LCOM1	0.6737		**0.0004046**	↗	**0.0009946**	↗
	LCOM2	0.9168		**0.001582**	↗	**0.0017**	↗
	LCOM5	0.6976		**0.0083**	↗	**0.001383**	↗
Complexity	McCabe	0.8881		**0.001085**	↗	**0.00063**	↗
	SIX	0.6163		**0.002085**	↗	**0.0008183**	↗
	WMC1	0.4008		**0.01341**	↗	**0.0467**	↗
	WMC	0.9252		**0.0003315**	↗	**0.001297**	↗
Coupling	ACMIC	0.896		0.066		**0.04961**	↗
	ACAIC	0.6251		0.8771		0.5029	
	CBO	0.513		**0.000176**	↗	**0.001948**	↗
	CBOin	0.8466		**0.0001986**	↗	**0.0005939**	↗
	CBOout	0.6496		**9.21E-06**	↗	**0.0001025**	↗
	connectivity	0.1912		0.8574		0.2603	
	CP	0.4471		**0.03687**	↗	0.1428	
	DCAEC	**0.002435**	↗	0.2118		0.06724	
	DCC	0.3617		**2.05E-05**	↗	**0.002347**	↗
	DCMEC	0.06408		0.2239		0.595	
	RFP	0.9383		0.6465		0.6074	
	RRFP	0.6811		0.993		0.5106	
	RRTP	0.8693		0.6973		0.6952	
	RTP	0.8923		0.4358		0.3693	
Inheritence	AID	0.6621		**0.03946**	↗	0.1391	
	ANA	0.1938		0.5803		0.3918	
	CLD	**0.002887**	↗	0.9954		**0.003298**	↘
	DIT	0.3		**0.02209**	↗	0.2632	
	NCM	0.6426		**0.008635**	↗	0.07486	
	NOA	0.9256		**0.01207**	↗	**0.01153**	↗
	NOC	**0.003547**	↗	0.1792		0.245	
	NOD	**0.0002031**	↗	0.166		0.07	
	NOH	0.7356		0.7807		0.9663	
	NOP	0.4834		**0.0008245**	↗	**0.007146**	↗
	ICHClass	0.8911		**0.000905**	↗	**0.001095**	↗
Issues	Numbers of Issues	0.0728		0.1603		0.6645	
Size and Polymorphism	CIS	0.4914		0.05132		0.1605	
	DAM	0.6724		**0.03264**	↗	**0.003362**	↗
	DSC	0.5031		0.4196		0.8725	
	EIC	0.6013		0.4277		0.5616	
	EIP	0.1874		0.5998		0.1039	
	MFA	0.9776		0.2374		0.243	
	MOA	0.9682		**0.01269**	↗	**0.01493**	↗
	NAD	0.921		**0.00277**	↗	**0.003884**	↗
	NADExtended	0.8652		**0.008383**	↗	**0.005466**	↗
	NMD	0.4008		**0.01341**	↗	**0.0467**	↗
	NCP	0.7092		0.4407		0.1198	
	NMA	0.3501		0.1012		0.3157	
	NMDExtended	0.5107		**0.02384**	↗	0.05112	
	NMI	0.4371		**0.02397**	↗	0.2016	
	NMO	0.8188		**0.0006559**	↗	**0.0005408**	↗
	NOM	0.4008		**0.01341**	↗	**0.0467**	↗
	NOParam	0.8372		0.1123		0.1551	
	NOPM	0.2496		0.9574		0.2793	
	PIIR	0.588		0.7276		0.2846	
	PP	0.8226		0.1112		0.1468	
	REIP	0.87		0.4993		0.3336	
	RPII	0.4809		0.652		0.8614	
Ranking	Class Rank	0.2598		0.9978		0.212	

5.6 Threats to Validity

We now discuss the threats to validity of our study following the guidelines provided for case study research [Yin, 2002].

Construct validity threats in this study concern our definition of motif composition; there is actually no agreed-upon definition of motif composition. We defined a motif composition as the implementation of two different roles in two different motifs by a same class. We only considered pairs of roles and ignored the effect of the particular roles on a class. We also explicitly excluded auto-composition, *i.e.*, a class playing two different roles in a same motif. Future work should distinguish compositions based on their roles and further study auto-compositions. Also, we purposefully studied only main roles of design motifs. Future work includes extending our study to all roles.

Internal validity threats in this study concern our inference. Our approach relies on the precision of the automatic detection approach DeMIMA. The results include false positives. We try to limit the number of false positives through a manual validation. However, the manual validation is a tedious task that leads to resilience and the experimenter bias: some false positives class may pass the validation because it "looks like" a motif. An approach that would provide a better precision is to use a manually-validated repository of motifs such as PMARt [Guéhéneuc *et al.*, 2004]. However, PMARt does not contain enough data as of now to perform such a study. We used as a baseline for our study of classes playing 1-role and 2-role, the 0-role population of classes playing none of the 11 roles considered in our study. However, among these classes, some may be playing one or two roles in *other* design motifs. Future work should extend this study to cover the 23 patterns from Gamma *et al.* [1994]

Threats to **external validity** concern the possibility to generalise our results. We studied six systems of different sizes, domains, maturity, and complexity. However, these systems are all open-source systems written in Java. We choose six design patterns among the many available. The results could be different with industrial systems, other object-oriented programming languages, and different design motifs.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. Moreover, both Eclipse source code repository and issue-tracking system are available to obtain the same data. Finally, the data from which our statistics have been computed is available on-line[1].

[1] `http://khomh.net/experiments/thesis/`

Conclusion validity threats concern the relation between the treatment and the outcome. There is no threat to the validity of the conclusion of this study as there is a direct relation between the chosen metrics and the overall internal quality of a class.

5.7 Summary

In this chapter, we presented a study of the impact of playing one or two roles in some motif(s) for a class. We answered the following research questions:

RQ1. In average, 8.24% (respectively 17.81%) of the classes of the six studied systems played one role (respectively two roles) in some motifs. These percentages are not negligible and therefore justify *a posteriori* the interest in design motif identification and *a priori* future studies on the impact of motifs on the quality of systems.

RQ2. Despite the few numbers of classes displaying a relationship between roles, we can conclude that some roles are more often played in pairs than others, for example (Decorator.Decorator, State.State). Further studies must focus on this research question to bring further generalisable evidence.

RQ3. There is a significant increase in many metric values, in particular for classes playing two roles. This increase confirms *a posteriori* the warning addressed to the community by Bieman, Beck, and others on the use of design patterns.

RQ4. There is a significant increase in the frequencies and numbers of changes of classes playing two roles. We thus confirmed on new samples the previous results by Di Penta *et al.*

We show that developers should be wary of classes playing two roles because they have significantly higher complexity metric values and represent 56% of changes while 1-role classes only 33%.

Globally, a particular attention should be paid to classes playing roles, in particular 2-role classes, because they have internal and external metric values that are significantly higher than these of other classes: they are more change-prone, less cohesive, more coupled, more complex, and more issue-prone.

Chapter 6

Antipatterns and Quality of Systems

Antipatterns—such as those presented in [Brown *et al.*, 1998]—have been proposed to embody poor design choices; Brown's 40 antipatterns describe the most common pitfalls in the software industry. These antipatterns stem from experienced developers' expertise and are conjectured in the literature to negatively impact systems quality [Brown *et al.*, 1998]. They are opposite to design patterns [Gamma *et al.*, 1994] and are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Despite the many studies on antipatterns summarised in Section 2.3 of Chapter 2, only a few studies empirically analysed the impact of antipatterns on source code-related phenomena [Bois *et al.*, 2006 ; Wei and Raed, 2007], in particular classes change and fault-proneness, which are the quality attributes of interest in this dissertation.

In practice, antipatterns are in-between design and implementation: they concern the design of one or more classes, but they concretely manifest themselves in the source code as classes with specific code smells [Fowler, 1999]. Often, antipatterns are defined in terms of thresholds imposed on metric values [Moha *et al.*, 2008a ; Moha *et al.*, 2008b]. In the preliminary studies presented in Chapter 3, we found that code smells significantly impact the change proneness of classes, therefore we expect antipatterns to have a similar impact.

One example of an antipattern is the LazyClass, shown in Listing 6.1. It occurs when a class does too little, *i.e.*, has few responsibilities in a system. A LazyClass is revealed by a class with few methods and fields; its methods have little complexity. A LazyClass often stems from speculative generality during a system design and–or implementation.

```
1  RULE_CARD : LazyClass {
2    RULE : LazyClass { INTER NotComplexClass FewMethods };
3    RULE : NotComplexClass { (METRIC: WMC, VERY_LOW, 20) };
4    RULE : FewMethods { (METRIC: NMD + NAD, VERY_LOW ,5) };
5  };
```

Listing 6.1 – Specification of the LazyClass Antipattern.

A more complex example of antipattern is the Blob, shown in Listing 6.2. A Blob, also called God Class, is a large and complex class that centralises the behaviour of a portion of a system and only uses other classes as data holders, *i.e.*, data classes. A Blob prevents the use of polymorphism through inheritance, making changes more complex and risk-prone. A class is a Blob if it has a low cohesion, it is large, some of its method names recall procedural programming, and, it is associated to data classes. Data classes only provide fields and–or accessors to their fields.

```
1  RULE_CARD : Blob {
2     RULE : Blob { ASSOC: associated FROM: mainClass ONE TO: DataClass MANY };
3     RULE : mainClass { UNION LargeClassLowCohesion ControllerClass };
4     RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion };
5     RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH, 0) };
6     RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH, 20) };
7     RULE : ControllerClass { UNION
8               (SEMANTIC: METHODNAME, {Process, Control, Ctrl, Command, Cmd,
9                                       Proc, UI, Manage, Drive})
10              (SEMANTIC: CLASSNAME,  {Process, Control, Ctrl, Command, Cmd,
11                                      Proc, UI, Manage, Drive, System, Subsystem}) };
12    RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90) };
13 };
```

Listing 6.2 – Specification of the Blob Antipattern.

Following our method DEQUALITE presented in Chapter 1, we quantify the impact of antipatterns on the change- and fault-proneness of classes using data mined from version control systems and issues reporting systems. More specifically, we study whether classes participating in an antipattern have an increased likelihood to change or to be involved in issues documenting faults than other classes. These results are important to build our quality models presented in Chapter 8.

6.1 Context

The context of this study consists in the change history and issues reporting systems of four Java systems: ArgoUML, Eclipse, Mylyn, and Rhino. The four systems have different sizes and belong to different domains. Eclipse is a large system (release 3.3.1 is larger than 3.5 MLOCs) and, therefore, close to the size of many real industrial systems. ArgoUML, Mylyn, and Rhino have wide ranges of sizes, are open-source, and have been the

Table 6.1 – Summary of the characteristics of the analysed systems. (The column Fault-fixing Changes report the number of *issue*-fixing changes in the case of Eclipse).

Systems	Releases (#)		Classes	LOCs	Changes	Fault-fixing Changes
ArgoUML	0.10.1–0.26.2	(10)	792–1,841	128,585–316,971	40,409	2,064
Eclipse	1.0–3.3.1	(13)	4,647–17,167	781,480–3,756,164	196,193	34,634
Mylyn	1.0.1–3.1.1	(18)	1,625–2,762	207,436–276,401	36,328	118
Rhino	1.4R3–1.6R6	(13)	89–270	30,748–79,406	6,925	1,068

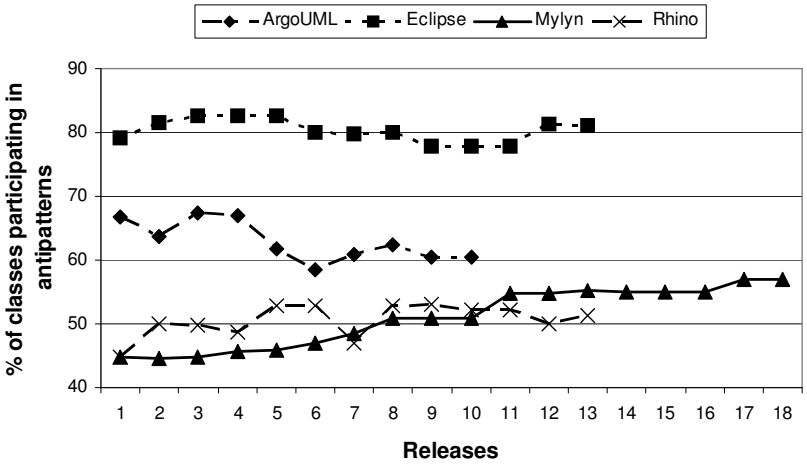


Figure 6.1 – Percentages of classes participating in antipatterns in the releases of the four systems.

subjects of previous studies. In particular, Eaddy *et al.* [2008] manually validated faults in Mylyn and Rhino. Section 4.5 presents their description and Table 6.1 summarises the main characteristics of the systems (detailed figures are available in Appendix C; fault classification for Mylyn is only available for the first three releases [Eaddy *et al.*, 2008]).

For the four systems, it is relevant to study the relation between antipatterns, change and fault-proneness, and class size, because the percentages of classes participating in antipatterns are not negligible. Figure 6.1 shows that the percentages of classes participating in antipatterns varies across releases of the four systems. The percentages of classes participating in antipatterns are higher for Eclipse ($\sim 80\%$) and ArgoUML ($\sim 60\%$) than for Mylyn and Rhino ($\sim 50\%$). With some variations, the percentages tend to remain stable for Eclipse and Rhino, to slightly decrease (from 67% to 60%) for ArgoUML, and to increase (from 45% to 57%) for Mylyn. Table 6.2 shows the distribution of the antipat-

Table 6.2 – Distribution of antipatterns in the analysed releases.

Antipatterns	Number of Antipatterns 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)

terns of interest, detailed in Appendix C. It also shows, in parentheses, the percentages of classes participating in each of the studied antipatterns in the first and last release. Percentages go as high as 51.71% of classes participating in LazyClass in the first version of Eclipse.

6.2 Study Definition and Design

The *goal* of this chapter is to investigate the relation between classes participating in antipatterns and their change- and fault-proneness as well as the kinds of changes impacting antipatterns. Our *purpose* is to bring generalisable, quantitative evidence on the impact of antipatterns on class change- and fault-proneness and build quality models. The *quality focus* is the quality of systems, specifically, source code change- and fault-proneness, that, if high, can have a concrete effect on developers' effort and on the overall project development and maintenance cost and time.

The *perspective* is that of researchers and practitioners, interested in the relation between antipatterns and evolution phenomena in software systems to build quality models and to understand and forecast the characteristics of classes. Developers who perform development or maintenance activities will also benefit from this study; they will be able to take into account and forecast their effort. Testers will increase their knowledge on classes that are important to test. Finally, the results of this study will also be of interest for managers and–or quality assurance personnel, who could use antipattern detection

techniques to assess the future changes and faults of in-house or to-be-acquired source code to better quantify its cost-of-ownership.

6.2.1 Research Questions

In this chapter, we address six null hypotheses, specifically concerning the relations between classes participating in antipatterns and their change-proneness (RQ1 and RQ2), fault-proneness (RQ3 and RQ4), size (RQ5), and kinds of changes (RQ6).

- **RQ1:** *What is the relation between antipatterns and change-proneness?* We investigate whether classes participating in at least one antipattern are more change-prone than others, by testing the null hypothesis: H_{01}*: the proportion of classes undergoing at least one change between two releases is not different between classes in antipatterns or not.*

- **RQ2:** *What is the relation between kinds of antipatterns and change-proneness?* We analyse whether certain antipatterns imply more changes than others, by testing the null hypothesis: H_{02}*: classes participating in certain antipatterns are not more change-prone than others.*

- **RQ3:** *What is the relation between antipatterns and fault-proneness?* This research question focuses on the relation between antipatterns and fault-fixing issues. The null hypothesis is: H_{03}*: the proportion of classes undergoing at least one fault-fixing between two releases does not differ between classes participating or not in at least one antipattern.*

- **RQ4:** *What is the relation between particular kinds of antipatterns and fault-proneness?* We also analyse the influence of kinds of antipatterns on fault-proneness, by testing the null hypothesis: H_{04}*: classes participating in certain kinds of antipatterns are not more prone to fault-fixing than other classes.*

- **RQ5:** *Do the presence of antipatterns in classes relate to the sizes of these classes?* This research question stems from El Emam *et al.* [2001] findings showing that many metrics correlate to size. Specifically, we study whether the higher change- and–or fault-proneness of classes participating in antipatterns is due to the sizes (in terms of LOC) of these classes or to the presence of the antipatterns, by testing the hypothesis: H_{05}*: classes participating in antipatterns are not larger than other classes.*

- **RQ6:** *What kind of changes are performed on classes participating or not in antipatterns?* We study whether classes participating in antipatterns undergo more (or less) structural changes (addition/removal/change of/to attributes, addition/removal of methods, or changes to the method signatures) than other kinds of changes by testing the hypothesis: H_{06}*: classes participating in antipatterns do not undergo a number of structural changes different than other kinds of changes.*

Hypotheses H_{01} to H_{05} are one-tailed, because we are interested in investigating only whether antipatterns relate to an *increase* of change-proneness, fault-proneness, and size. Hypothesis H_{06} is two-tailed because we investigate whether the presence of antipatterns is related to a higher or a lower number of structural changes.

6.2.2 Independent Variables

We use the approach, DECOR (Defect dEtection for CORrection) presented in the Section 4.2, to specify and detect the following antipatterns: AntiSingleton, Blob, ClassDataShouldBePrivate (CDSBP), ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList (LPL), MessageChains, RefusedParentBequest (RPB), SpaghettiCode, SpeculativeGenerality (SG), SwissArmyKnife.

We choose these antipatterns only because (1) they are well-described by Brown [1998], (2) we could find enough of their occurrences in several releases of several of the studied systems, and (3) they are representative of design and implementation problems with data, complexity, size, and the features provided by classes.

Our independent variables are the number of classes participating in the 13 antipatterns. Variables $ap_{i,j,k}$, indicate the numbers of times that a class i participates in an antipattern j in a release k. For RQ1 and RQ3, we aggregate these variables into a Boolean variable $AP_{i,k}$ indicating if a class i participates or not in any antipattern.

6.2.3 Dependent Variables

Dependent variables measure the phenomena related to classes participating in antipatterns.

RQ1 and RQ2. For each class of our studied systems, we compute the change-proneness as defined in Section 4.4.

RQ3 and RQ4. We compute the fault-proneness of classes, as discussed in Section 4.4.

RQ5. The size of classes participating or not in antipatterns is measured using their LOCs, excluding comments and blank lines. Each classes is associated with its size, the total number of antipatterns and the kind of antipatterns in which it participates. (Abstract and native methods and methods declared in interfaces count for zero LOC.)

RQ6. Kinds of changes are computed for classes participating in antipatterns following the definition presented in Section 4.4; we determine different kinds of changes performed on a class c_i participating in an antipattern, by comparing the class revision in releases k and $k+1$. At least one difference between $c_{i,k}$ and $c_{i,k+1}$ indicates that c_i, and thus the antipattern in which it participates, has been changed *w.r.t.* release k.

6.2.4 Analysis Method

RQ1 and RQ3. We study whether changes to and faults/issues in a class are related to the class participating in antipatterns, regardless of the kinds of antipatterns. Therefore, we test whether the proportions of classes exhibiting (or not) at least one change/fault/issue significantly vary between classes participating in antipatterns and other classes. We use Fisher's exact test (presented in Section 4.6.1) for H_{01} and H_{03}. (We did not consider releases with a number of changes/fault/issues lower than 10 because Fisher's test would not be applicable). We also compute the *odds ratio* (OR) (see Section 4.6.1).

RQ2 and RQ4. We want to understand the relation of specific kinds of antipatterns with changes and faults/issues. Let us focus on RQ2 and changes. We use a logistic regression model [Hosmer and Lemeshow, 2000] to test H_{02} and H_{04} by correlating the presence of antipatterns with changes. Details on the logistic regression model are presented in Section 4.6.5.

We count, for each antipattern, the number of times that, across the analysed releases, the p-values obtained by the logistic regression are significant. We use the current state of the art threshold $t = 75\%$ [Conte and Campbell, 1989 ; Vicinanza *et al.*, 1991] to assess whether classes participating to a specific kind of antipattern have significantly greater odds to change than others: If these classes are more likely to change in more than t releases, then this antipattern has a significant negative impact on change-proneness.

RQ5. We perform the analysis related to RQ5 in three steps.

First, we compare, for each release, the average size of (1) classes participating in at least one antipattern and (2) classes participating in no antipattern. We use the Wilcoxon test and compute Cohen *d* effect size (see Section 4.6). We expect the test results to be statistically significant and the odds ratios to be greater or equal to 1 because many antipatterns are, according to their definitions, related to size, *eg.*, Blob, ComplexClass, and LargeClass.

Second, we perform the same test and compute the same odds ratios between the set of classes participating in each antipattern and those not participating in any antipattern. We expect that, for some antipatterns, the test would not be significant and–or the odds ratios would be lower than 1. Indeed, while the definitions of some antipatterns directly relate to their size, others specifically target small classes, *eg.*, LazyClass, or are orthogonal to size, *eg.*, ClassDataShouldBePrivate.

Third, we again perform Fisher's exact test and compute the odds ratios between large classes participating or not to size-related antipatterns, *i.e.*, Blob, ComplexClass, and LargeClass. We single out classes whose sizes are greater than the 75% percentile and divide them in two sets: those participating in the considered antipatterns and those that do not participate in these antipatterns. We expect that classes participating in Blob, ComplexClass, and LargeClass antipatterns are not significantly larger than the largest classes.

RQ6. We again use Fisher's exact test to compare the proportions of structural changes in classes participating in antipatterns with those of other kinds of changes, also in classes not participating in any antipattern.

6.3 Study Results

This section reports the results of our empirical study, which are further discussed in Section 6.4. Detailed results are presented in Appendix C while raw data is available on-line[1].

[1] `http://khomh.net/experiments/thesis/`

Table 6.3 – Summary of the odds ratios for classes that participate in at least one antipattern to underwent at least one change.

Change Proneness							
ArgoUML		Eclipse		Mylyn		Rhino	
Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios
0.10.1	4.17	**1.0**	1.13	1.0.1	10.51	1.4R3	10.41
0.12	7.16	2.0	**0.75**	2.0M1	10.37	1.5R1	17.98
0.14	6.22	2.1.1	2.59	2.0M2	7.38	1.5R2	17.37
0.16	15.84	2.1.2	1.42	2.0M3	206.60	1.5R3	15.71
0.18.1	10.00	2.1.3	1.15	2.0	14.17	1.5R4	16.19
0.20	26.54	3.0	**0.88**	2.1	10.89	1.5R41	30.71
0.22	8.83	3.0.1	**0.86**	2.2.0	11.10	1.5R5	15.51
0.24	15.40	3.0.2	**0.89**	2.3.0	9.83	1.6R1	24.73
0.26	3.98	3.2	2.19	2.3.1	7.66	1.6R2	12.69
0.26.2	6.75	3.2.1	1.94	2.3.2	24.38	1.6R3	19.95
		3.2.2	1.47	3.0.0	9.45	1.6R4	33.05
		3.3	2.43	3.0.1	9.85	1.6R5	19.97
		3.3.1	1.42	3.0.2	5.31	1.6R6	20.56
				3.0.3	8.18		
				3.0.4	3.77		
				3.0.5	4.96		
				3.1.0	10.53		
				3.1.1	5.59		

6.3.1 RQ1: What is the relation between antipatterns and change-proneness?

Table 6.3 summarises the odds ratios when testing H_{01}. In all releases, except Eclipse 1.0, Fisher's exact test indicated a significant difference (p-values < 0.05) of proportions between change-prone classes among those participating and not in antipatterns.

Odds ratios vary across systems and, within each system, across releases. While in few cases, ORs are close to 1, *i.e.*, the odds is even that a class participating in an antipattern changes or not, in some pairs of systems/releases, such as ArgoUML 0.20, Mylyn 2.0M3, or Rhino 1.5R41, ORs are greater than 25. Overall, ORs for Eclipse are lower than those of other systems, by one or two orders of magnitude. The odds of a class participating in some antipatterns to change are, in general, higher than that of other classes.

We therefore conclude that, in general, *there is a relation between antipatterns and change-proneness*: a greater proportion of classes participating in antipatterns change

Table 6.4 – Summary of odds ratios for classes that participate in at least one antipattern to underwent at least one fault/issue fixing.

Fault/Issue Proneness							
ArgoUML		Eclipse		Mylyn		Rhino	
Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios
0.10.1	4.43	1.0	1.32	1.0.1	10.45	1.4R3	6.44
0.12	4.87	2.0	1.57	2.0M1	17.70	1.5R1	31.29
0.14	17.53	2.1.1	1.70	2.0M2	>>300	1.5R2	–
0.16	6.58	2.1.2	2.00	2.0M3	–	1.5R3	13.93
0.18.1	5.33	2.1.3	2.03	2.0	–	1.5R4	9.06
0.20	4.95	3.0	2.52	2.1	–	1.5R41	30.05
0.22	9.42	3.0.1	1.95	2.2.0	–	1.5R5	10.57
0.24	2.25	3.0.2	1.86	2.3.0	–	1.6R1	29.26
0.26	8.08	3.2	2.72	2.3.1	–	1.6R2	–
0.26.2	9.73	3.2.1	2.19	2.3.2	–	1.6R3	–
		3.2.2	2.05	3.0.0	–	1.6R4	23.00
		3.3	3.18	3.0.1	–	1.6R5	13.29
		3.3.1	1.23	3.0.2	–	1.6R6	–
				3.0.3	–		
				3.0.4	–		
				3.0.5	–		
				3.1.0	–		
				3.1.1	–		

with respect to other classes. The rejection of H_{01} and the ORs provide *a posteriori* concrete evidence of the negative impact of antipatterns on change-proneness.

6.3.2 RQ2: What is the relation between kinds of antipatterns and change-proneness?

Table 6.5 summarises the results of the logistic regression for the relations between change-proneness and the different kinds of antipatterns. A cell in the table reports the number of releases of a given system (per column) in which classes being a given antipattern (per row) correlate to change-proneness with statistical significance. For example, the cell at the intersection of the column for ArgoUML and the row for AntiSingleton reports that, in 8 releases of ArgoUML out of 10 (80%), classes participating in the AntiSingleton were more change-prone than other classes significantly.

From Table 6.5, we can reject H_{02} for some antipatterns, *i.e.*, for antipatterns which are significantly correlated to change-proneness in at least $t = 75\%$ of the releases, highlighted

Table 6.5 – Summary of the number and percentages of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	Change Proneness			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	**8 (80%)**	5 (38%)	7 (39%)	–
Blob	2 (20%)	8 (62%)	9 (50%)	–
CDSBP	3 (30%)	7 (54%)	9 (50%)	6 (46%)
ComplexClass	2 (20%)	**12 (92%)**	2 (11%)	–
LargeClass	2 (20%)	–	4 (22%)	4 (31%)
LazyClass	5 (50%)	**12 (92%)**	3 (17%)	1 (8%)
LongMethod	**10 (100%)**	**12 (92%)**	**17 (94%)**	5 (38%)
LPL	**9 (90%)**	**10 (77%)**	7 (39%)	3 (23%)
MessageChain	**10 (100%)**	**12 (92%)**	**18 (100%)**	**13 (100%)**
RPB	**9 (90%)**	6 (46%)	10 (56%)	5 (38%)
SpaghettiCode	–	–	–	–
SG	–	3 (23%)	6 (33%)	1 (8%)
SwissArmyKnife	–	6 (46%)	–	–

in gray. Following our analysis method, only MessageChain has a significant negative impact on change-proneness in all systems: classes participating in this antipattern are more likely to change than classes participating in other or no antipattern in more than $t = 75\%$ of the releases. Other antipatterns have significant impact on a subset of the systems: LongMethod in ArgoUML, Eclipse, and Mylyn; LongParameterList in ArgoUML and Eclipse; AntiSingleton and RefusedParentBequest in ArgoUML; Complexclass and LazyClass in Eclipse.

We conclude that *there is a relation between kinds of antipatterns and change-proneness* but not for all antipatterns and not consistently across systems and releases.

6.3.3 RQ3: What is the relation between antipatterns and fault-proneness?

Table 6.4 summarises Fisher's exact test results and ORs for H_{03}. The differences in proportions are significant and thus we can reject H_{03} in all cases. The proportion of classes participating in antipatterns and reported in faults is between 1.32 and 31.29 times larger than that of other classes.

Odds ratios for faults are not always higher than those for changes: although classes participating in antipatterns are more likely to exhibit fault fixing changes than other classes, they seem to be even much more likely to undergo restructuring changes in addition to fault-fixing changes than other classes with better design.

Table 6.6 – Summary of the number and percentages of significant p-values across the analysed releases obtained by logistic regression for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	Fault/Issue Proneness			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	5 (50%)	**13 (100%)**	–	–
Blob	1 (10%)	7 (54%)	–	–
CDSBP	2 (20%)	7 (54%)	2 (66%)	3 (33%)
ComplexClass	–	**13 (100%)**	1 (33%)	–
LargeClass 3 (30%)	–	–	3 (33%)	
LazyClass	–	**12 (92%)**	–	2 (22%)
LongMethod	1 (10%)	**13 (100%)**	–	3 (33%)
LPL	5 (50%)	9 (60%)	2 (66%)	3 (33%)
MessageChain	7 (70%)	**10 (77%)**	1 (33%)	**7 (78%)**
RPB	4 (40%)	4 (31%)	1 (33%)	–
SpaghettiCode	–	–	–	–
SG	–	4 (31%)	–	1 (11%)
SwissArmyKnife	–	1 (8%)	–	–

Therefore, we conclude that *there is a relation between antipatterns and fault-proneness*; although this relation is not as strong as the relation with change-proneness.

6.3.4 RQ4: What is the relation between particular kinds of antipatterns and fault-proneness?

Table 6.6 reports the results of the logistic regression for the relations between fault/issue-proneness and kinds of antipatterns. For Mylyn, we could analyse only 3 releases for fault-proneness and for Rhino, only 9 releases, because of the limited number of faults occurring in other releases (< 10). We can reject H_{04} for MessageChain in Eclipse and Rhino; AntiSingleton, ComplexClass, LazyClass, and LongMethod in Eclipse.

We conclude that *there is a relation between kinds of antipatterns and fault/issue-proneness* but not for all antipatterns and not consistently across systems and releases.

6.3.5 RQ5: Do the presence of antipatterns in classes relate to the sizes of these classes?

We found that, as expected, classes participating in some specific kinds of antipatterns are significantly larger (with a *medium* to *large* effect size) than classes not participating in antipatterns, with the following exceptions:

- Classes participating in AntiSingleton are not significantly larger than classes not participating in any antipattern in 10 out of 18 Mylyn releases;
- Classes participating in LazyClass are significantly smaller than other classes in all the analysed releases of ArgoUML, Mylyn, and Rhino. This observation was expected because, by definition, LazyClasses are small;
- Classes participating in RefusedParentBequest are not significantly larger than classes not participating in any antipattern in 1 out of 10 ArgoUML releases, 15 out of 18 Mylyn releases, and 9 out of 13 Rhino releases;
- Classes participating in SpeculativeGenerality are not significantly larger than classes not participating in any antipattern in 5 out of 10 ArgoUML releases, all 18 Mylyn releases, and all 13 Rhino releases.

In Eclipse, all kinds of antipatterns classes have a significantly larger size than classes not participating in any antipattern, although for the above-mentioned antipatterns the effect size was generally *small* while for the others it was *medium* to *small*.

Table 6.7 – ORs of change- and fault-proneness for large classes participating in the Blob, LargeClass, ComplexClass antipatterns *w.r.t.* large classes not participating in any antipattern (bold face indicates statistical significance of the Fisher's exact test).

ArgoUML			Eclipse			Mylyn			Rhino		
Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)
0.10.1	–	–	1.0	0.98	2.12	1.0.1	0.92	1.09	1.4R3	–	–
0.12	–	0.79	2.0	0.73	1.03	2.0M1	**5.60**	–	1.5R1	–	–
0.14	**7.81**	–	2.1.1	1.79	1.14	2.0M2	1.46	–	1.5R2	5.76	–
0.16	–	3.41	2.1.2	1.78	2.56	2.0M3	–		1.5R3	5.51	17.72
0.18.1	2.32	3.05	2.1.3	1.58	1.02	2.0.0	**2.56**		1.5R4	–	–
0.20	–	0.45	3.0	1.22	0.81	2.1	1.17		1.5R41	–	–
0.22	2.64	0.57	3.0.1	0.96	1.55	2.2.0	2.85		1.5R5	3.10	**13.52**
0.24	5.26	–	3.0.2	0.93	0.99	2.3.0	0.77		1.6R1	1.67	**13.52**
0.26.2	**3.46**	1.85	3.2	**2.18**	**5.81**	2.3.1	**3.89**		1.6R2	8.53	–
0.26	**3.10**	1.32	3.2.1	**2.56**	**2.84**	2.3.2	3.95		1.6R3	2.30	–
			3.2.2	**1.52**	**1.99**	3.0.0	**5.39**		1.6R4	–	–
			3.3	**8.08**	**13.66**	3.0.1	1.16		1.6R5	3.63	4.19
			3.3.1	**2.48**	**1.57**	3.0.2	**2.89**		1.6R6	–	–
						3.0.3	1.04				
						3.0.4	1.66				
						3.0.5	2.38				
						3.1.0	–				
						3.1.1	**3.27**				

Finally, Table 6.7 reports results (ORs, highlighted in bold face when the test reported statistical significance) of the Fisher's exact test, comparing change and fault-proneness

of classes having a size greater than the 75% percentile of the overall size distribution and participating or not in the Blob, ComplexClass, and LargeClass antipatterns. Although the test only reports statistical significance in a small number of cases—due to the limited number of classes having a size above the 75% percentile of the distribution—ORs are greater to 1 in:

1. ArgoUML: 6 out of 10 releases for change-proneness and 4 out of 7 releases for fault-proneness;
2. Eclipse: 9 out of 13 releases for change-proneness and 9 out of 13 releases for fault-proneness (plus 2 other cases where the ORs are just above one);
3. Mylyn: 13 out of 15 releases for change-proneness (plus another case where the ORs is just above one), while it was not possible to get statistically-significant results for fault-proneness, due to the limited number of detected occurrences;
4. Rhino: 7 out of 8 releases for change-proneness and 4 out of 5 releases for fault-proneness.

ORs are always above one, and in most cases above two and up to 13.66 every time the Fisher's exact test found a statistically-significant difference in the proportions of change- and fault-prone between (large) classes participating or not to size-related antipatterns. Therefore, large classes participating in antipatterns change more and are more fault-prone than large classes not participating to any antipatterns.

We conclude that classes participating in antipatterns are generally larger than other classes. This conclusion was expected because, many antipatterns, such as Blob, ComplexClass, LargeClass, or LongMethod, result from an excessively large size *and* of other negative characteristics of the classes. We also conclude that, except for some releases of the analysed systems, some kinds of antipatterns (AntiSingleton, LazyClass, RefusedParentBequest, and SpeculativeGenerality) describe symptoms of poor design that are unrelated to size.

We thus generally conclude that *some kinds of antipatterns are related to size as expected by their definitions but size only does not explain the classes greater change- and fault-proneness.*

Table 6.8 – Fisher's exact test results and odds ratios of the proportions of all kinds of changes to classes participating in antipatterns *w.r.t.* those of other classes.

Systems	p-values	ORs
ArgoUML	< 0.01	1.22
Eclipse	< 0.01	1.03
Mylyn	< 0.01	1.19
Rhino	0.08	1.04

6.3.6 RQ6: What kind of changes are performed on classes participating or not in antipatterns?

While studying the relation between kinds of antipatterns and change- and fault-proneness, we also studied the kinds of changes impacting classes participating in antipatterns.

Results are reported in Table 6.8. For simplicity's sake, and because we did not notice substantial changes across releases, we report results obtained by aggregating data from the whole observed history of each system, rather than for each release separately.

Table 6.8 shows that classes participating in antipatterns in Rhino do not undergo more structural changes than other changes: it is a small system and, therefore, the different kinds of changes may occur to any class. Although we can reject H_{06} for Eclipse, the OR ≈ 1 downplays this result, which we explain by the use of inheritance in Eclipse [Aversano *et al.*, 2007], leading to few structural changes to classes.

Detailed results for different kinds of antipatterns reveal that, for all antipatterns except LazyClass in ArgoUML, Mylyn, and Rhino, and RefusedParentBequest in Eclipse, classes participating in antipatterns undergo more structural changes than others changes (*eg.*, changes in the method implementations). The methods implementations of LazyClasses, as reported in Section 6.3.2, change to increase their behaviour. Changes in the methods organisations and implementations of RefusedParentBequest are generally performed to correct them.

We conclude that *structural changes occur more often on classes belonging to antipatterns than other changes.*

6.4 Discussions

We now discuss the results using the releases' histories. Roman superscript numerals are reported in Table 6.9. We also discuss the impact of the class sizes on our results. Finally,

we discuss our results in relation to previous results from Chapters 3 and 5 relating change- and fault-proneness with code smells and design patterns.

6.4.1 Correlations among Antipatterns

We analysed whether there exists a correlation between the presence of different antipatterns and, hence, between their definitions. We used the non-parametric Spearman correlation, which results indicate that Blob, ComplexClass, and LargeClass are lowly correlated ($0.5 < \rho < 0.7$ [Cohen, 1988]) in all releases of ArgoUML, Mylyn, and Rhino, but in none of Eclipse. For all other antipatterns and releases, we obtained no correlation among antipatterns ($\rho \ll 0.5$). We expected that we would not find correlations among antipatterns, because their definitions are different as they capture different types of design pitfalls. We also analysed whether we could find a correlation between the presence of different antipatterns and traditional object-oriented metrics, such as Chidamber and Kemerer's metric suite. As we expected since antipatterns are higher level than metrics, we could not find a correlation that was consistent across systems and their releases, thus showing that antipatterns, albeit detected using metrics, bring different information to developers than metrics.

6.4.2 Statistical Significance/Unexpected Ratios

Tables 6.3 and 6.4 show that, in general, classes belonging to antipatterns are more change- and fault-prone than others. However, there is a case were H_{01} could not be rejected for lack of statistical significance and four cases with unexpected ORs, which indicate that classes participating in antipatterns changed less than others (shown in grey in the table).

We explain the lack of statistical significance for Eclipse 1.0 by the major changes between releases 1.0 and 2.0[i], which imply that many classes were added/changed (Eclipse size increased from 781 to 1,250 KLOCs and 4,647 to 6,742 classes), irrespective of their participation in antipatterns.

The first case with an unexpected OR concerns classes having changed between Eclipse 2.0 and 2.1.1, with $OR = 0.75$. Eclipse 2.1[iii] introduced several new features *w.r.t.* 2.0, including navigation history, sticky hovers, prominent status indication, and so on. Moreover, 283 issues[iv] were fixed between 2.0 and 2.1 and 126 more[v] between 2.1 and 2.1.1, including issues related to the new features, for example issue ID 1694 *"FEATURE: Contributed inspection formatter"* or 17872 *"Hover help for static final fields is inconsistent"*. Finally, $8,730 - 6,742 = 1,988$ classes were added for an increase of $1,797 - 1,2490 = 548$

Table 6.9 – URLs of the discussed release notes and issues listings .

IDs	Systems	URLs
i	Eclipse	http://archive.eclipse.org/eclipse/downloads/drops/R-2.0-200206271835/buildNotes.php
ii	Mylyn	http://eclipse.org/mylyn/new/new-3.0.html
iii	Eclipse	http://archive.eclipse.org/eclipse/downloads/drops/R-2.1-200303272130/whats-new-all.html
iv	Eclipse	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=2.1&resolution=FIXED&order=bugs.bug_id
v	Eclipse	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=2.1.1&resolution=FIXED&order=bugs.bug_id
vi	Eclipse	http://www.eclipse.org/osgi/
vii	Eclipse	For example, a search for "Eclipse 3.0 crash" returns 224 messages on http://www.eclipsezone.com/
viii	Eclipse	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.0.1&resolution=FIXED&order=bugs.bug_id
ix	Eclipse	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.0.2&resolution=FIXED&order=bugs.bug_id
x	Eclipse	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.2&resolution=FIXED&order=bugs.bug_id
xi	ArgoUML	http://argouml.tigris.org/servlets/NewsItemView?newsItemID=1675
xii	ArgoUML	http://argouml.tigris.org/servlets/NewsItemView?newsItemID=830
xiii	Mylyn	http://eclipse.org/mylyn/new/new-2.0.html
xiv	Rhino	http://www.mozilla.org/rhino/rhino15R1.html
xv	Rhino	https://bugzilla.mozilla.org/buglist.cgi?query_format=specific&order=relevance+desc&bug_status=__all__&product=Rhino&content={1.6R3|1.6R4|1.6R5}

KLOCs. Such dramatic changes explain the odds ratio, as many classes not belonging to antipatterns were added/changed.

The second, third, and fourth cases concern classes having changed between releases 3.0 and 3.2. Eclipse 3.0 was a major improvement over the 2.x series, with a new runtime platform implementing the OSGi R3.0 specifications[vi] to become a Rich Client Platform to develop any tools (not necessarily an IDE). It had many problems at first[vii], corrected in the subsequent 3.0.1, 3.0.2, and 3.2 releases, with respectively 266[viii], 70[ix], and 285 issues[x] fixed. No less than $15,153 - 11,166 = 3,987$ classes were added between 3.0 and 3.2, which did not only belong to antipatterns. Eclipse size increased by $3,271 - 2,260 = 1,011$ KLOCs.

6.4.3 Changes/Faults Odds Ratios

For ArgoUML, change-proneness ORs are never smaller than 3.98. The highest OR occurs between releases 0.20 and 0.22, period during which a major restructuring[xi] took place with many faults fixed and 293 issues resolved. ORs for fault-fixing are high but often lower

than those for change-proneness, which suggests that antipatterns are potential symptoms of change-proneness, but not necessarily of fault-proneness: they make a system harder to maintain because future changes will likely impact several classes, but only indirectly impact fault-proneness. The highest fault-related OR occurs between releases 0.14 and 0.16, period during which many fault-fixing activities took place[xii]. Release 0.16 is the release with the highest number of fault-fixing changes: 851, the second-highest is release 0.26 with 591 (see Appendix C).

For Eclipse, we found lower ORs than those of other systems for both class change- and fault-proneness. We explain such a difference by the fact that $\sim 80\%$ of Eclipse classes participate in at least one antipattern, with a higher proportion of these classes to be LazyClasses (*eg.*, 51.71% in the first release). Therefore, we expected to find lower ORs because Eclipse includes many more classes participating in antipatterns than not. The high proportion of LazyClasses is conform to the results of previous studies [Aversano *et al.*, 2007], which observed that Eclipse is designed to evolve through sub-classing, which, in turn, leads to a lower class change-proneness.

Eclipse is the only system with greater ORs for fault/issue- than change-proneness. We recall that we considered issues, as discussed in Section 6.2.3, and that as discussed in our previous study [Antoniol *et al.*, 2008], a majority of Eclipse issues are likely *not* related to faults but to other maintenance activities, such as restructuring and enhancement. Thus, it is consistent to find more classes impacted by issues *w.r.t.* faults only.

Verifying H_{01} for Mylyn between releases 2.0M3 and 2.0 results in an extreme OR = 206.60, which we explain by the amount of issues fixed between the releases: 304[xiii]. Table 6.4 shows that antipatterns are correlated with fault-fixing changes. The OR reflects this relation plus that with other changes unrelated to faults, such as restructuring, which impacted classes in antipatterns.

For Rhino, ORs for change-proneness range between 10.41 in release 1.4R3 and 33.05 in 1.6R4, two numbers which we explain by (1) the number of new features added in release 1.5R1[xiv]: many classes not participating in antipatterns were added/changed and (2) the number of issues between releases 1.6R3, 1.6R4, and 1.6R5: respectively 4, 7, and 24[xv]. More faults have been filled against 1.6R4 than other releases, thus explaining the change of ORs.

6.4.4 Kinds of Antipatterns and Changes/Faults

Tables 6.5 and 6.6 show that antipatterns impact change- and fault-proneness but that we could not reject H_{02} or H_{04} for all of them, in particular LargeClass, Blob, ClassDataShouldBePrivate, SpaghettiCode, SpeculativeGenerality, and SwissArmyKnife. We explain this fact by the low number of classes participating in these antipatterns: on average, in Eclipse, there are 479 LargeClasses for 11,618 classes per release; in ArgoUML, 80 for 960 classes per release; and so on (see Appendix C). The number of SpaghettiCode is even lower, with, on average, 2 per Eclipse release. No SpaghettiCode was found in ArgoUML, Mylyn, and Rhino.

Eclipse, ArgoUML, Mylyn and Rhino use extensively object orientation. They "divide to conquer", which helps to avoid: Blob, which is a class that knows/does too much; LargeClass and SwissArmyKnife, which are complex classes that provides too many services; and SpeculativeGenerality, which is an abstract class with very few children. The use of polymorphism and encapsulation explains the few number of ClassDataShouldBePrivate, which occurs when the data encapsulated by a class is public, as well as the SpaghettiCode, which is a class with too many long methods with too many branches.

Moreover, classes with these antipatterns appear to remain unchanged or to be removed from the systems as they evolve. For example, in Eclipse 2.0, the only LargeClass was `org.eclipse.core.internal.indexing.IndexedStoreException`, which changed only once between 2.0 and 2.1.1 and never changed again. Only one SwissArmyKnife was present in Mylyn 1.0.1, which was removed in 2.0.0, no SwissArmyKnife was found in the other studied versions.

Among the remaining antipatterns, MessageChain enables the rejection of H_{02} for all systems and H_{04} for Eclipse and Rhino. This antipattern characterises classes that use long chains of calls to perform their functionality, which makes them dependent on classes far from each other. Finding many MessageChain is not surprising in Eclipse and Rhino. Eclipse has thousands of classes; developers fixing issues are likely to touch many classes because of their relations with one another and the risk of faults caused by these relations is high. Rhino is small but the classes forming its parse tree and interpreter are tightly coupled.

The other antipatterns satisfy the conditions to reject H_{02} or H_{04} for at least one system. By following their presence through releases, we found that antipatterns are generally removed from the system while some new ones are introduced. Thus, some

antipatterns are in small number or are absent in some releases, and the logistic regression analysis indicated that some antipatterns are statistically significant only in some releases.

Classes participating in the antipatterns ComplexClass and LazyClass are more change- and fault-prone than others in Eclipse. ComplexClass characterises classes with a higher number of methods than the average, thus developers adding new features or fixing issues are more likely to touch these classes, which consequently increases their risks to have faults. This observation confirms Fowler and Brown's warnings about complex classes. Lazy classes tend to be removed, or changed to increase their behaviour, while others are introduced: there were 2,765 lazy classes in Eclipse 1.0 (59% of the system), 8,967 in 3.3.1 (52%). Class `org.eclipse.search.internal.core.SearchScope`, for example, was a lazy class in 1.0 but, in 3.0, 2 methods and 2 constructors were added and the inner class `WorkbenchScope` was removed. New lazy classes, *eg.*, `org.eclipse.team.internal.ccvs.ui.actions.ShowEditorsAction`, were introduced.

Classes participating in AntiSingleton are more fault/issue-prone in Eclipse and more change-prone in ArgoUML than other classes. They are generally removed from the system or changed. In Eclipse 16% of the AntiSingleton classes were removed between releases 1.0 and 3.0 and only 53% of the classes were still AntiSingleton in that release; the other classes were changed. For example, all methods of `org.eclipse.compare.internal.CompareWithEditionAction`, an AntiSingleton, were removed between releases 1.0 and 3.0 and the class became a LazyClass with no behaviour.

We can reject H_{02} for LongMethod for ArgoUML, Eclipse, and Mylyn, and H_{04} for Eclipse. LongMethod classes are more change-prone than any other class, and more fault-prone than other Eclipse classes, possibly because such classes are complex and thus more likely to change to fix issues. Faults are also more likely to be introduced when changing these classes due to their complexity. Moreover, we observe that LongMethod classes keep on participating in this antipattern during their evolution and are, in general, central to the system core features. Previous studies, *eg.*, [Aversano *et al.*, 2007], confirm that central classes are more change-prone.

Classes participating in RefusedParentBequest are more change-prone than others in ArgoUML, possibly due to the need for re-organising badly organised hierarchies: this antipattern occurs when a subclass does not use attributes and–or public/protected methods inherited from its parent. We expected this results because ArgoUML implements deep hierarchies of models, diagram elements, and tools.

6.4.5 Development Practice and Antipatterns

The results of this chapter show that antipatterns do impact the change- and fault-proneness of classes negatively and that certain kinds of antipatterns have a greater impact than others. However, we do not claim that antipatterns *cause* changes and faults. Indeed, our study cannot say anything about the *reasons* for classes to have antipatterns and, consequently, the reasons for changes and faults to appear in these classes. We only empirically verified that classes with antipatterns are more change- and fault-prone than others, thus confirming the conjecture in the literature.

In addition, in some situations, an antipattern may actually be the best and possibly only way to implement some requirements and–or functionalities of the systems. For example, one of the LargeClass in Eclipse is class `org.eclipse.swt.internal.win32.OS`, which is the unique access point for the Standard Widget Toolkit to the underlying Windows platform. Although the class is large, it is sensible to provide a unique access point to non-object-oriented, platform-dependent resources, thus increasing portability and possibly efficiency.

Another important issue about the potential usefulness of antipatterns is whether they provide more information than size. El Emam *et al.* [2001] found that many metrics are correlated to size, thus antipatterns could also be correlated to size because they use metrics. However, as discussed in RQ5 (see Section 6.3.5), we found that this is not the case for many antipatterns, *i.e.*, classes participating to some kinds of antipatterns are not significantly larger than other classes. Moreover, large classes participating in antipatterns are generally more change- and fault-prone than other large classes.

In Section 3.2, we showed that classes having code smells—the symptoms of antipatterns—are more change-prone than classes having no code smells. These results are confirmed and reinforced the results reported in this chapter. Indeed, considering that code smells make classes more change-prone, it is natural that classes having antipatterns are also more change-prone than classes without antipatterns. However, this study was necessary to confirm this relation because we did not test in our preliminary study the different impact on a class to have one, two, or more particular code smells. It could have been possible that having some combinations of code smells could have rendered classes more difficult to change and, consequently, less change-prone than others.

We can also relate the impact of having two or more antipatterns on change- and fault-proneness with the impact of playing two or more roles in design motifs studied in Chapter 5. We used several metrics, including the number of past and future changes

and the number of faults, to study the impact of playing roles. We showed, using a representative population of classes, that classes playing one, two, or more roles are more change prone than classes playing zero roles. We could not find any statistically significant impact of playing one role vs. two roles on change- and fault-proneness but classes playing two roles changed 1.52 times more than classes playing one role. Classes participating in one or more antipatterns are somehow similar to those playing two or more roles in some design motifs because, in both cases, they are bigger than others and play either a central role in the functioning of the system (design motifs) or in the maintenance of the system (antipatterns). In Chapter 8, we combine our findings on these design motifs and antipatterns to build quality models to help characterize change and fault prone classes.

6.5 Threats to Validity

We now discuss the threats to validity of our study.

Construct validity threats in this study are mainly due to measurement errors. The identification of changes is reliable because based on the CVS/SVN change logs. It may not reflect exactly the commits related to a change/fault-fixing and developers' efforts accurately because developers follow different patterns for committing their changes, *eg.*, from committing changes as faults are fixed to committing all changes once a week. However, this does not affect our measure of change-proneness as we are interested whether a class underwent at least one change during a given period of time.

We were able to identify fault-fixing changes for ArgoUML, Mylyn, and Rhino using an existing classification [Eaddy *et al.*, 2008]. For Eclipse, we related antipatterns with issue-proneness; we are aware that issue-fixing does not equal fault-fixing. We focus on issues marked as "FIXED" or "CLOSED" because they required changes. It is unlikely that hard-to-fix issues would stay longer "OPENED" than others in Eclipse, because Eclipse is being backed up by IBM, which strives to offer a stable product.

Finally, we observe that DECOR includes its authors' subjective understanding of the antipatterns and that the accuracy of its detection algorithms is not perfect [Moha *et al.*, 2009]. DECOR accuracy impacts our results because we may have classified a class not participating in an antipattern as participating in it and vice-versa. Other techniques and tools should be used to confirm our findings. However, we found that classes with these antipatterns as detected by DECOR are more change- and fault-prone than other classes, therefore detecting antipatterns with DECOR can possibly help developers to focus their effort.

Threats to **internal validity** do not affect this study, being an exploratory study [Yin, 2002]. Thus, we cannot claim causation, but relate the presence of antipatterns with the occurrences of changes, faults, and issues. Nevertheless, we tried to explain—by looking at specific changes, commit notes, and change histories—why some antipatterns could have been the cause of changes/issues/faults. We are aware that antipatterns can be dependent to each other and relied on the logistic regression model-building procedure to select the subset of non-correlated antipatterns. When studying antipatterns, we do not exclude that, in a particular context, an antipattern can be the best way to implement or design a (part of a) system.

Threats to **external validity** concern the possibility to generalise our results. First, we studied four systems having different size and belonging to different domains. Nevertheless, further validation on a larger set of systems is desirable. Second, we used a particular yet representative subset of antipatterns. Different antipatterns could have led to different results and should be studied in future work. Within its limits, our results confirm the conjectures reported in the literature.

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. The raw data used to compute the statistics is available on-line[1].

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 normality of the data set distribution. For Mylyn, we are aware that fault-proneness is analysed on only 3 releases for which the manual fault classification is available [Eaddy *et al.*, 2008], thus it would be difficult to make strong conclusions, for this system, about the relation between antipatterns and fault-proneness.

6.6 Summary

In this chapter, we provided empirical evidence of the negative impact of antipatterns on classes change- and fault-proneness in four systems: ArgoUML, Eclipse, Mylyn, and Rhino. We studied the odds ratios of changes, faults, and issues on classes participating (or not) in 13 antipatterns in (overall) 54 releases of the four systems. We showed (see **RQ1**–**5**) that classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes (issue-fixing changes for Eclipse)

than other classes. We also showed that size alone cannot explain the participation of classes to antipatterns (**RQ6**) and, thus, that antipatterns bring additional, complementary information to developers to analyse their systems.

We also studied the kinds of changes that impacted classes participating in antipatterns and other classes. We found that, in ArgoUML, and Mylyn, structural changes are more likely to occur in classes participating in antipatterns, although odds ratios are not high (≈ 1.2), than in other classes, while it is not the case for Eclipse and Rhino. Furthermore, we studied the correlations among antipatterns and found that the Blob, ComplexClass, and LargeClass are correlated with one another in all releases of ArgoUML, Mylyn, and Rhino, but in none of Eclipse. As expected, other antipatterns are unrelated.

This study provides evidence to practitioners that they should pay attention to systems with a high number of classes participating to antipatterns, because these classes are more likely to contain faults and to be the subject of their change efforts. More specifically, managers and developers can use these results to monitor the quality of systems and guide maintenance activities: for example, they can recommend their developers to avoid MessageChain as this antipattern is consistently related with high fault and change rates.

Chapter 7

Relation between Antipatterns and Design Patterns

Design patterns are *good* solutions to recurring design problems while antipatterns are *poor* solutions. In Chapters 6 and 5, we investigated independently the impact of antipatterns and of design patterns on quality.

In a previous study [Vaucher *et al.*, 2009], we analysed the lifecycle of God Classes in Xerces and Eclipse JDT and found that some God Classes interact with Abstract Factory, Adapter, Observer, and Prototype. This result drew our attention on the possible co-occurrence of antipatterns and design motifs in object oriented systems.

Theoretically, antipatterns and design patterns could be argued to be unrelated by definition. However, our results in Chapter 5 showed that design patterns do not always impact positively the quality characteristics of systems and of their classes, *eg.*, we found that tangled design pattern implementations make classes more complex and less cohesive. Other studies [Aversano *et al.*, 2007 ; Di Penta *et al.*, 2008] reported that some design patterns and their roles are more resilient to changes than others and thus contribute to increase a class change-proneness. In our first preliminary study, presented in Chapter 3, we also found that developers think that some design patterns decrease system extensibility.

On the contrary, antipatterns, could be the best way to design and implement part of a system in some specific cases, as we remarked in Chapter 6 (p. 100). For example, one of the LargeClass in Eclipse is class `org.eclipse.swt.internal.win32.OS`, which constitutes the unique access point for the Standard Widget Toolkit to the underlying Windows platform and which provides a unique access point to non-object-oriented, platform-dependent resources, contributes in increasing portability and possibly efficiency.

Table 7.1 – Characteristics of the systems.

Systems	Releases (#)		Classes	LOCs	Changes
ArgoUML	0.10.1–0.22	(9)	792–1,619	128,585–279,864	29,906
Eclipse-JDT	1.0–3.0	(5)	1,382–2,518	257,605–528,522	67,165
Mylyn	2.0.0–3.1.1	(13)	1,759–2,647	185,169–276,401	22,804
Rhino	1.4R3–1.6R7	(14)	89–270	30,748–79,406	6,925

In this chapter, we perform a systematic study of the co-occurrence of several design patterns and antipatterns in multiple software systems and analyzed the effects of such a co-occurrence. For consistency and to emphasise the parallel between design patterns and antipatterns, we use the term *antipatterns* (AP) to refer to antipatterns and code smell, and *design pattern* (DP) to refer to design motifs [Guéhéneuc and Antoniol, 2008]: ideal solutions that describe the roles played by classes to implement the motifs. Thus, we say that a class *participate* in some APs (respectively, DPs) when having some code smells (respectively, playing some roles).

Understanding the interactions between APs and DPs is important for the accuracy of quality models build from them. Indeed, the majority of statistical techniques generally used to build models assumes an independence of observations *i.e.*, knowledge of the value of one observation should not provide information about the value of any other. A violation of this assumption generally results in poor accuracy, incorrectly narrow confidence limits and incorrectly small p values. Consequently, possible confounding effects of APs and DPs interactions should be control.

7.1 Context

The context of this study consists of the following four open-source systems: ArgoUML, Eclipse-JDT, Mylyn, and Rhino. The systems have different sizes and belong to different domains. Section 4.5 presents their description and Table 7.1 reports some descriptive statistics of the four systems. Since changes (as detailed in Chapter 4) are computed between a release and the subsequent one, the last release for which we relate the presence of AP and DP with change-proneness is—for the four systems—the release preceeding the one reported as last release in Table 7.1, namely release 0.20 for ArgoUML, 2.1.3 for Eclipse-JDT, 3.1.0 for Mylyn, and 1.6R6 for Rhino. Table 7.2 briefly describes the studied APs and DPs and reports the minimum and maximum numbers of their occurrences in each system.

Table 7.2 – Design patterns/antipatterns considered in the study and the minimum and maximum numbers of their occurrences in releases of the four systems.

Names	When a class participates in the antipattern/design pattern explanation	Numbers of Occurrences (Min–Max)			
		ArgoUML	Eclipse-JDT	Mylyn	Rhino
Antipatterns					
AntiSingleton	Provides mutable class variables, which could be used as global variables	242–420	99–254	9–127	1–16
Blob	Too large, not enough cohesive, monopolises most of the system processings	12–84	131–362	48–93	—
ClassDataShouldBePrivate (CDSBP)	Exposes its fields, thus violating the principle of encapsulation	43–158	174–214	55–183	4–33
ComplexClass	Has (at least) one large and complex method	0–94	167–333	34–72	6–14
LargeClass	Has (at least) one large method, in term of lines of code	0–144	—	52–99	9–19
LazyClass	Has few methods and fields; its methods have little complexity	16–1202	719–1323	4–71	4–11
LongMethod	Has a method that is overly long	172–259	864–1581	153–349	14–35
LongParameterList (LPL)	Has (at least) one method with an overly-long list of parameters	0–252	426–1300	39–95	8–28
MessageChain	Uses a long chain of method calls to realise (at least) one of its functionality	0–260	332–669	75–181	20–66
RefusedParentBequest (RPB)	Redefines inherited method using empty bodies, breaking polymorphism	0–403	116–301	65–290	5–16
SpeculativeGenerality (SG)	Defined as abstract; has a very few children, which do not use its methods	0–15	9–12	14–39	0–6
SwissArmyKnife	Its methods provide many different unrelated functionalities	—	23–25	—	—
Design Patterns					
Adapter (A)	Allows classes with different interfaces to interact	492–1183	1023–1789	474–1014	61–138
Command (Cmd)	Encapsulates a request as an object	271–601	779–1331	129–368	36–89
Composite (C)	Lets clients treat individual objects and compositions of objects uniformly	220–664	122–344	82–218	35–65
Decorator (D)	Dynamically attaches additional responsibilities to an object	108–203	355–542	25–120	31–64
Factory Method (FM)	Defines an API for object creation; subclasses choose the class to instantiate	142–450	713–997	91–342	27–68
Observer (O)	Dynamically defines a one-to-many dependency between objects	128–384	207–278	40–60	30–61
Prototype (P)	Specifies the kind of objects to create using a prototypical instance	0–46	52–107	0–145	—
State (S)	Allows an object to alter its behaviour when its internal state changes	0–873	741–1302	197–589	50–97
Template Method (TM)	Allows subclasses to define a varying behaviour	0–1074	913–1581	422–926	49–116
Visitor (V)	Separates an algorithms from the structure on which it operates	0–23	256–439	0–214	0–32

7.2 Study Definition and Design

Our *goal* in this chapter is to study classes participating in APs and DPs with the *purpose* of investigating whether classes participating in APs also participate in DPs and the change-proneness of such co-occurrences. We do not study the fault-proneness because the available data set does not allow to analyse combinations of APs and DPs without risk of Type II error (*i.e.*, error of failing to observe a difference when in truth there is one). Indeed, data on faults are available only for few releases of some of the systems. The *quality focus* is related to the quality of systems and to the change proneness of classes participating in APs and DPs. The *perspective* is that of researchers and practitioners

who would benefit of knowledge on the impact of co-occurrences of APs and DPs on class change-proneness in software systems to build quality models.

7.2.1 Research Questions

In this chapter, we aim at answering the following research questions:

- **RQ1:** *What is the number of classes participating in antipatterns and design patterns?*
- **RQ2:** *What is the impact on change-proneness for a class to participate both in some antipatterns and design patterns?*
- **RQ3:** *What is the impact of playing roles in particular kinds of antipatterns* and *design patterns w.r.t. change-proneness?*

7.2.2 Independent Variables

We again use DECOR and DeMIMA presented in Chapter 4 to specify and detect APs and DPs presented in Table 7.2. Our independent variables are the number of classes participating in these APs and DPs.

7.2.3 Dependent Variables

The dependent variable of this study is class change-proneness, computed as discussed in Section 4.4.

7.2.4 Analysis Method

We now present methods used to answer our research questions.

RQ1: What is the number of classes participating in antipatterns and design patterns? This question is preliminary to the following ones. It provides quantitative data on the percentages with which APs and DPs occur and co-occur in the releases of the studied systems.

We address this question by analysing whether some classes, in all studied releases, participate to APs and DPs. Specifically, we show for each releases (1) the overall num-

ber of classes, (2) the number of classes participating in APs (3) the number of classes participating in DPs, (4) the number of classes participating both in APs and DPs.

Then, we perform a deeper analysis about the co-occurrences of APs and DPs, reporting the number and percentages of pairs of AP code smells and DP roles that have the highest percentages of co-occurrence in the four systems. We limit our analysis to percentages above 10% of the overall number of classes (5% for Mylyn and Rhino, due to their smaller numbers of classes participating in APs).

RQ2: What is the impact on change-proneness for a class to participate both in some antipatterns and design patterns? We investigate if the co-occurrence of APs and DPs has a positive or negative effect, *i.e.*, if it is related to higher or lower change-proneness *w.r.t.* APs and DPs alone.

We first analyse to what extent changes to a class are related to its participation in at least one AP and one DP, regardless of the kind of AP and DP. Therefore, we test the null hypothesis[1] H_{02}: *the proportion of classes undergoing at least one change between two subsequent releases does not change between all classes participating in some APs and classes participating in both some APs and DPs.*

We use Fisher's exact test (presented in Section 4.6.1) and also compute the *odds ratio* (OR) (see Section 4.6.1). Specifically, we compute OR considering (1) as experimental group the classes participating in APs and as control group classes not participating in any AP and (2) as experimental group classes participating in both some APs and DPs and as control group again classes not participating in any AP. $OR = 1$ indicates that the event is equally likely in both samples; $OR > 1$ that the event is more likely in the experimental group while $OR < 1$ indicates that it is more likely in the control group.

RQ3: What is the impact of playing roles in particular kinds of antipatterns *and* design patterns *w.r.t.* change-proneness? We explore whether some kinds of DPs impact more some kinds of APs *w.r.t.* class change-proneness. If the co-occurrence of a DP decreases the odds to change of a class participating in an AP, then we say that the DP "loves" this AP, else that it "hates" it.

We investigate whether specific APs and–or DPs influence class change-proneness and whether the interaction between APs and DPs is significantly correlated to a higher class change-proneness. For each pair of AP–DP, we test the null hypothesis H_{03}: *the partici-*

[1]There is no H_{01} because RQ1 is exploratory.

Table 7.3 – Number and percentage of classes (across all releases) participating at least once in antipatterns and–or design patterns.

Systems	Classes	Classes APs		Classes DPs		Classes APs+DPs	
ArgoUML	2,834	1,791	(63%)	1,998	(71%)	1,650	(58%)
Eclipse-JDT	3,144	2,709	(86%)	2,495	(79%)	2,141	(68%)
Mylyn	3,437	1,229	(36%)	2,346	(68%)	1,102	(32%)
Rhino	560	160	(29%)	397	(71%)	154	(28%)

pation of a class in DP does not interact with its participation in an AP w.r.t. increasing or decreasing its change-proneness.

We use a logistic regression model (see Section 4.6.5) to correlate the participation of classes in some APs and DPs with changes. Like in Chapter 6, we use it as an alternative to the Analysis Of Variance (ANOVA) to test the influence of the participation of a class in APs and DPs, and the interaction of the participation to both APs and DPs, to the class change-proneness.

Ideally, to test interactions between APs and DPs, we should include as independent variables of the logistic regression model all APs code smells and DPs roles and their pair-wise interactions. However, due to the high number of possible combination and the relatively small data set, such inclusion would not produce any statistically significant result. Therefore, we follow three steps: (1) we build a logistic regression model without considering interactions between APs and DPs. We identify with this model those APs and DPs having a significant effect on change-proneness; (2) we build a model with the independent effect and the interaction of all APs and DPs identified in Step (1); and, (3) for all significant interactions, we compute the change-proneness OR (*w.r.t.* other classes) for classes participating in the APs, DPs, and both, to identify the "love" and "hate" relationships.

7.3 Study Results

This section reports the study results. Raw data for replication are available on-line[2].

[2] `http://khomh.net/experiments/thesis/`

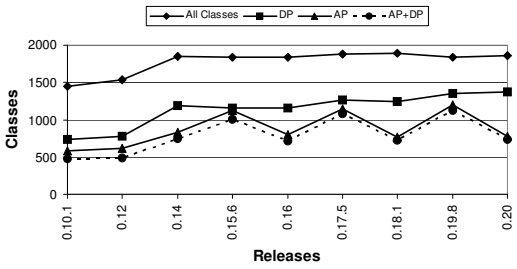
7.3.1 RQ1: What is the number of classes participating in antipatterns and design patterns?

Table 7.3 shows the overall number of classes participating (at least once in the observed time frame) in APs and–or DPs across all releases of the four systems and their percentages. It shows that (1) percentages of classes participating in APs vary between system: from 86% in Eclipse-JDT to 29% in Rhino; (2) all four systems contain a high percentage of classes participating in DPs: from 68% in Mylyn to 79% in Eclipse-JDT; (3) percentages of classes participating in both APs and DPs are, again, higher for Eclipse-JDT (68%) and ArgoUML (58%), and lower for Mylyn (32%) and Rhino (28%). In summary, these figures indicate that the percentages of classes participating in APs *and* DPs is worth further investigation.

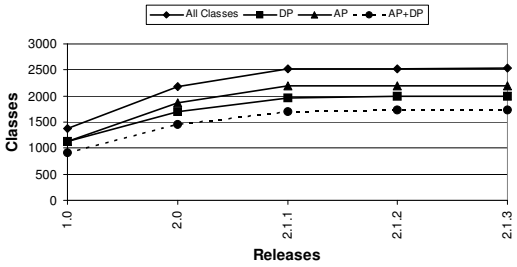
Figure 7.1 shows the evolution of the number of classes and of the numbers of classes participating in APs and–or DPs: (1) for ArgoUML (in Figure 7.0(a)), the number of classes slightly increases between releases 0.12 and 0.14 and then remains stable. A similar phenomenon occur for classes in DPs. The number of classes participating in APs (and similarly, that of classes in APs and DPs) exhibits a periodic behaviour between releases 0.14 and 0.20; (2) for Eclipse-JDT (in Figure 7.0(b)), the number of classes and of classes participating in APs, DPs, and in both, exhibit similar behaviours, *i.e.*, they increase until release 2.1.1 and then tend to remain stable. Differently to the three other systems, the number of classes participating in APs is always higher than that of classes in DPs; (3) for Mylyn (in Figure 7.0(c)), again, the series evolve consistently, with a small increase in correspondence of release 3.0.0 and 3.1.0; and, (4) for Rhino (in Figure 7.0(d)), the number of classes participating in APs as well as those in both APs and DPs—is low and stable, similarly to that of classes in DPs.

Table 7.4 shows the top 10 pairs of APs and DPs with the highest percentages of co-occurrences, limited to the pairs in which the number of overlapping classes is $> 10\%$ (respectively, $> 5\%$) of the total number of classes of ArgoUML and Eclipse-JDT (respectively, of Mylyn and Rhino). It shows that: (1) for ArgoUML, 10 DP roles largely co-occur with LazyClass ($\leq$77% of the occurrences); (2) for Eclipse-JDT, we observe frequent occurrences of DPs with LongMethod; (3) for Mylyn, percentages of co-occurrence are lower, *i.e.*, 36% and below. DPs highly co-occur with LongMethod (as in Eclipse-JDT), RefusedParentBequest, and MessageChain; and, (4) for Rhino, DPs largely co-occur with MessageChains.

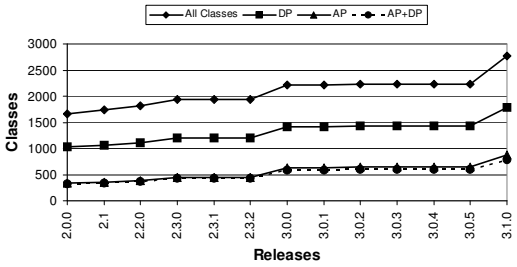
(a) ArgoUML



(b) Eclipse-JDT



(c) Mylyn



(d) Rhino

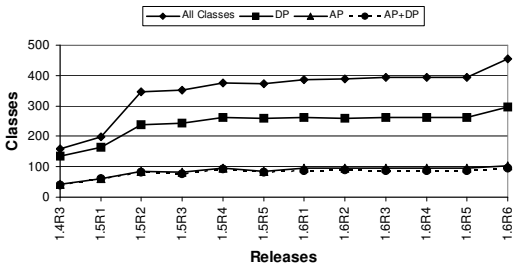


Figure 7.1 – Evolution of the numbers of classes participating in APs, DPs, and APs+DPs.

Table 7.4 – Highest percentages of co-occurrences between APs and DPs.

Design Patterns Roles	Antipatterns Code Smells	# of DPs	# of APs	Perc.
ArgoUML				
C.Leaf	LazyClass	702	661	94%
O.ConcreteObserver	LazyClass	404	379	94%
TM.Client	LazyClass	1347	1141	85%
A.Client	LazyClass	1284	1085	85%
A.Adapter	LazyClass	882	733	83%
S.Concretestate	LazyClass	1103	913	83%
A.Adaptee	LazyClass	1189	961	81%
TM.Concreteclass	LazyClass	1016	802	79%
FM.ConcreteProduct	LazyClass	443	337	76%
Cmd.Concretecommand	LazyClass	434	327	75%
Eclipse-JDT				
Visitor.ObjectStructure	LongMethod	367	343	93%
Cmd.invoker	LongMethod	620	568	92%
Visitor.Client	LongMethod	421	385	91%
S.Context	LongMethod	678	618	91%
FM.ConcreteCreator	LongMethod	532	473	89%
Cmd.Client	LongMethod	643	571	89%
D.Concretecomponent	LongMethod	455	391	86%
A.Adapter	LongMethod	1035	845	82%
FM.ConcreteProduct	LongMethod	764	623	82%
Cmd.Concretecommand	LongMethod	910	702	77%
Mylyn				
TM.Concreteclass	RPB	617	221	36%
A.Adapter	RPB	548	196	36%
A.Client	LongMethod	1024	334	33%
TM.Client	LongMethod	1071	343	32%
A.Adaptee	LongMethod	851	229	27%
A.Adaptee	RPB	851	221	26%
TM.Client	RPB	1071	266	25%
A.Client	MessageChain	1024	252	25%
A.Client	RPB	1024	249	24%
TM.Client	MessageChain	1071	254	24%
Rhino				
Visitor.Client	MessageChain	45	39	87%
Cmd.Client	MessageChain	38	31	82%
FM.ConcreteCreator	MessageChain	53	43	81%
O.ConcreteSubject	MessageChain	53	41	77%
O.ConcreteObserver	MessageChain	57	43	75%
FM.ConcreteProduct	MessageChain	52	37	71%
Cmd.invoker	MessageChain	63	44	70%
Cmd.Concretecommand	MessageChain	43	30	70%
C.Leaf	MessageChain	72	50	69%
D.Concretecomponent	MessageChain	72	50	69%

Table 7.5 – Change-proneness ORs for classes participating in APs and in both APs and DPs.

ArgoUML			Eclipse-JDT			Mylyn			Rhino		
Rel.	ORs APs	ORs APs+DPs	Rel.	ORs APs	ORs APs+DPs	Rel.	ORs APs	ORs APs+DPs	Rel.	ORs APs	ORs APs+DPs
0.10.1	14.17	2.08	1.0	1.42	1.50	2.0.0	14.17	9.16	1.4R3	10.41	7.12
0.12	7.16	1.91	2.0	0.72	0.62	2.1	10.89	5.82	1.5R1	17.98	11.37
0.14	5.36	2.33	2.1.1	2.46	2.81	2.2.0	11.10	6.68	1.5R2	17.37	15.00
0.15.6	97.44	22.78	2.1.2	0.89	0.98	2.3.0	9.83	5.52	1.5R3	15.71	8.63
0.16	15.91	4.47	2.1.3	1.88	1.91	2.3.1	7.66	4.61	1.5R4	27.04	16.07
0.17.5	19.81	5.09				2.3.2	24.38	14.95	1.5R5	15.51	8.55
0.18.1	8.60	4.01				3.0.0	9.45	5.94	1.6R1	24.73	13.87
0.19.8	11.45	3.72				3.0.1	9.85	5.63	1.6R2	12.69	9.53
0.20	26.54	12.27				3.0.2	5.31	3.41	1.6R3	19.95	15.85
						3.0.3	8.18	5.31	1.6R4	33.05	17.49
						3.0.4	3.77	2.27	1.6R5	19.97	13.98
						3.0.5	4.96	3.06	1.6R6	20.56	11.78
						3.1.0	10.53	8.39			

7.3.2 RQ2: What is the impact on change-proneness for a class to participate both in some antipatterns and design patterns?

Table 7.5 shows the ORs of classes participating in APs to change *w.r.t.* other classes (columns 2, 5, 8, 11) and the change-proneness ORs of classes participating in AP *and* DPs (columns 3, 6, 9, 12). We performed a Fisher's exact test, which indicates that *the proportion of change-prone classes participating in both APs and DPs is significantly higher than the proportion of change-prone classes participating to APs only* (p-value < 0.05), with the exception of classes in Eclipse-JDT releases 2.0 and 2.1.2. Thus, we reject in general H_{02}.

Classes participating in APs are more change-prone than other classes with ORs (much) higher than one. *Whenever an AP occurs in a class that also participates in a DP, the change-proneness OR always decreases.* The only exceptions to this phenomenon are in Eclipse-JDT, where the ORs for classes participating in APs are not very high and their ORs do not decrease (in some cases even slightly increase) when classes also participate in DPs.

Table 7.6 – Love–hate relationships: significant interactions among specific APs and DPs that decrease/increase the class change-proneness ORs.

Rel.	Design Patterns	Antipatterns	DPs OR	APs OR	Int. OR
Design Patterns "Love" Antipatterns					
ArgoUML					
0.14	S.Concretestate	Blob	7.32	55.01	0.09
0.14	A.Adapter	MessageChain	3.61	9.32	0.12
0.18.1	D.Concretecomponent	Antisingleton	1.29	8.68	0.09
0.18.1	A.Adaptee	LargeClass	6.54	25.15	0.23
0.18.1	FM.ConcreteCreator	MessageChain	9.91	11.12	0.14
Eclipse-JDT					
2.1.1	D.Concretecomponent	LongMethod	1.89	3.14	0.54
2.1.1	FM.ConcreteProduct	MessageChain	1.65	3.41	0.52
2.1.2	C.Leaf	LPL	1.02	1.07	0.38
2.1.3	FM.product	AntiSingleton	0.63	2.20	0.20
2.1.3	Cmd.Concretecommand	LPL	1.01	2.05	0.47
2.1.3	S.Concretestate	MessageChain	0.58	1.81	1.56
Mylyn					
2.3.0	FM.ConcreteCreator	LongMethod	4.21	17.58	0.07
2.3.1	Visitor.Client	LPL	16.84	24.49	0.04
3.0.3	S.Concretestate	CDSBP	3.22	5.63	0.18
3.0.3	S.Context	LongMethod	3.85	10.91	0.19
Design Patterns "Hate" Antipatterns					
ArgoUML					
0.14	Cmd.Concretecommand	RPB	3.74	1.60	11.02
Eclipse-JDT					
1.0	S.Context	LazyClass	2.70	1.17	5.54
1.0	FM.ConcreteProduct	LPL	1.95	1.18	4.53
2.0	Visitor.Client	MessageChain	1.04	0.59	19.02
2.1.1	S.Concretestate	ComplexClass	2.29	3.86	4.58
2.1.2	C.Leaf	ComplexClass	0.66	2.19	4.41
2.1.2	FM.ConcreteProduct	LazyClass	1.78	0.43	3.29
2.1.2	O.subject	LazyClass	3.32	0.66	5.43
2.1.2	S.Concretestate	LazyClass	1.48	0.39	2.38
2.1.2	Cmd.Concretecommand	LazyClass	1.60	0.47	1.74
2.1.2	Visitor.Client	LongMethod	0.84	2.01	6.97
2.1.2	O.subject	LPL	2.69	0.93	3.93
2.1.2	Visitor.Client	MessageChain	2.25	1.56	3.00
2.1.2	C.Leaf	MessageChain	0.39	2.30	4.31
2.1.3	S.Concretestate	AntiSingleton	0.78	1.66	1.93
2.1.3	FM.ConcreteCreator	CDSBP	1.97	0.53	2.46
2.1.3	C.Leaf	MessageChain	0.35	1.66	6.39
2.1.3	P.Concreteprototype	RPB	5.92	0.42	13.03

7.3.3 RQ3: What is the impact of playing roles in particular kinds of antipatterns *and* design patterns *w.r.t.* change-proneness?

We analyse the interactions between specific APs and DPs. The upper part of Table 7.6 shows the "love" relationships, *i.e.*, co-occurrences found in releases of the four systems for which the occurrence of a DP in a class participating in an AP causes a *decrease* of the class change-proneness OR. It shows the change-proneness ORs of classes participating (1) in the DP *w.r.t.* classes not participating in it, (2) in the AP *w.r.t.* classes not participating in it, and (3) in both the AP and the DP *w.r.t.* classes not participating in them. The table excludes Rhino because we could not obtain significant results due to the limited size of the data set.

Similarly, the bottom part of Table 7.6 shows the few "hate" relationships, *i.e.*, classes in which the co-occurrence of a specific DP with an AP causes an *increase* of the classes change-proneness ORs. Consistently with results of **RQ2**, "hate" relationships only occur in Eclipse-JDT and for one co-occurrence in ArgoUML.

7.4 Discussions

We now discuss the results to the research questions.

7.4.1 RQ1: What is the number of classes participating in antipatterns and design patterns?

Table 7.3 shows that large numbers of classes are found to participate in APs and DPs by the detection approaches. We discuss the results reported in Table 7.4 as follows:

- In ArgoUML, 58% of classes participate in some APs, DPs, and both (2^{nd} highest number of co-occurrences to Eclipse-JDT). A review of a sample of these classes suggest that DP-related methods are often long methods, with many parameters; for example the Adapter class *org.argouml.uml.cognitive.critics.ClClassName*, which method *panicking()* is also an occurrence of LongMethod and LongParameterList.

- In Eclipse-JDT, the co-occurrences with highest percentages are those with Long-Method. We expected such co-occurrences given the number of LongMethod in Eclipse reported in Table 7.2. Eclipse-JDT includes many LongMethods because of the intrinsic complexity of parsing/debugging/analysing Java source code. Also, a

large number of classes playing concrete-class or leaf roles in DPs are often LazyClasses, *eg.*, *org.eclipse.jdt.internal.corext.refactoring.changes.DeleteFolderChange* and *org.eclipse.jdt.internal.core.search.matching.MultipleSearchPattern* that are Decorator.ConcreteComponent, *org.eclipse.jdt.internal.debug.eval.ast.instructions.RemainderAssignmentOperator* that is a Composite.Leaf, and *org.eclipse.jdt.internal.core.builder.ClasspathLocation* that is a Visitor.ConcreteElement. We explain these results by the architecture of Eclipse-JDT, which is based on plug-ins, offering many extension points, and designed to evolve through sub-classing [Aversano *et al.*, 2007]: many classes are LazyClasses playing roles in DPs.

- In Mylyn, DPs highly co-occur with LongMethod, for reasons similar to those explained for Eclipse-JDT, with RefusedParentBequest because it subclasses many classes from the Eclipse platform and Eclipse-JDT plug-in. An example of RefusedParentBequest class is *org.eclipse.mylyn.internal.tasks.ui.TaskWorkingSetFilter* that is also a Composite.Leaf.

- In Rhino, MessageChain co-occurs with all DPs with the highest percentages. We explain the occurrence of MessageChain by the intrinsic construction of Rhino, which is a ECMA/JavaScript parser in which classes offer lots of proxy methods. These MessageChains are often Visitor and Command clients. In Rhino 1.6R6 for example, the class *org.mozilla.javascript.ImporterTopLevel* is a Visitor.Client, while *org.mozilla.javascript.JavaAdapter* is a Command.client.

Figure 7.1 shows the evolution of the numbers of classes participating in APs and–or DPs:

- In ArgoUML, the trend of the number of occurrences of APs follows a periodic behaviour. We explain such trend using the release notes. For example, between releases 0.19.8 and 0.20, the release notes[3] report that "Over 400 bugs have been fixed for this release." and "[a] lot of hard work has also happened under the covers". Similarly, between releases 0.14 and 0.15.6, a new version of GEF[4] has been integrated, which could explain the higher number of APs.

- Differently from the other three systems, in Eclipse-JDT, the number of classes participating in APs is always higher than the number of classes in DPs. We explain this observation by the high number of LongMethods, resulting from the implementation

[3] `http://argouml.tigris.org/servlets/NewsItemView?newsItemID=1451`
[4] `http://argouml.tigris.org/servlets/NewsItemView?newsItemID=732`

of Eclipse-JDT. These LongMethods are in the *org.eclipse.jdt.internal* packages, thus reflecting the complexity of implementing a Java IDE.

- In Mylyn and Rhino, the trends are as could be expected: the number of classes participating in APs and DPs slowly increase, consistently with the overall number of classes.

7.4.2 RQ2: What is the impact on change-proneness for a class to participate both in some antipatterns and design patterns?

The high ORs in columns labelled "APs" in Table 7.5 indicate that, in all releases of all systems but releases 2.0 and 2.1.2 of Eclipse-JDT, classes participating in APs are more change-prone than others. Eclipse-JDT 2.1 introduced several new features, including supports for building projects with circular dependencies, multi-JAR system libraries, source and output folders that are outside of the workspace. Some $2514 - 2181 = 333$ new classes were added, an increase of $528,314 - 438,070 = 90,244$ LOCs; 125 faults [5] were fixed between 2.0 and 2.1. and 53 between 2.1.2 and 2.1.3. Such dramatic changes explain the ORs: many classes not belonging to APs were added/changed.

Moreover, Table 7.5 also shows that, whenever APs and DPs co-occurs in a class, the change-proneness ORs always decreases except for the same two releases 2.0 and 2.1.2 of Eclipse-JDT. This decrease hints at *the potential capability of design patterns to mitigates the negative impact of APs on classes where these APs occur.*

7.4.3 RQ3: What is the impact of playing roles in particular kinds of antipatterns *and* design patterns *w.r.t.* change-proneness?

Table 7.6 summarises the "Love–Hate" relationships between specific APs and DPs. The "love" relationships shows that not all co-occurrences of some DPs in classes participating in APs have a negative impact on class change-proneness. Indeed, many of these co-occurrences significantly decrease the change-proneness ORs of the classes. This decrease suggests that, when (sub-)systems are properly designed and–or implemented using *good* solutions to recurring design problems, *i.e.*, DPs, then these DPs help mitigate and even revert the (sub-)systems decay into *poor* solutions, *i.e.*, APs.

The "hate" relationships only happens in Eclipse-JDT and for one co-occurrence in ArgoUML. A review of the classes involved in these "hate" relationships suggest that

[5] `http://archive.eclipse.org/eclipse/downloads/drops/R-2.1.3-200403101828/readme_eclipse_2_1_3.html\#KnownIssues`

they are in general the results of the decay of a DP, *eg.*, the unique "hate" relationship in ArgoUML, represented by class *org.argouml.uml.ui.foundation.core.UML Structural-FeatureMultiplicityCombo- BoxModel*, was a Cmd.Concretecommand in ArgoUML 0.14 and decayed into a RefusedParentBequest in ArgoUML 15.6 while still playing the role of Cmd.Concretecommand in subsequent releases. This decay caused an increase of the change-proneness OR of that class.

7.5 Threats to Validity

The results of any empirical studies are subject to the following threats to their validity.

Construct validity threats concern the relation between theory and observation: they are mainly due to measurement errors possibly introduced by the adopted APs and DPs detection approaches.

Precision and recall figures for DeMIMA and DECOR are consistent with the precision and recall of other approaches considering that all code smells/roles are identified. We encountered no scalability issue. Yet, as said before, DeMIMA and DECOR, similarly to other AP and DP detection approaches, include their authors' subjective understanding of the *intent* of the APs and DPs, and their translations into automated detection algorithms. However, we make no claim about the *actual* developers' intents or lack thereof in using these APs and DPs. Thus, a subjective understanding does not affect our study because we showed that occurrences and co-occurrences of APs and DPs—as defined in DeMIMA and DECOR—relate with class change-proneness, notwithstanding whether developers *intently* introduced them. Studying the developers' intent would require qualitative data that are currently unavailable and, thus, is part of future work.

Another threats to construct validity is our analysis of changes in classes that play a role in a DP, although these changes might be related to pieces of functionality that a class implements unrelated to the DP. The study reported in this paper is a coarse-grained analysis, investigating the relation between the participation of classes in APs and DPs, and changes occurring at class-level. Future work will aim at investigating change-proneness at a finer-grained level.

Threats to **internal validity** do not affect this study, being an exploratory study [Yin, 2002]. Thus, we cannot claim causation but report, in **RQ2** and **RQ3**, the relation between occurrences of APs and–or DPs and class change-proneness. Nevertheless, we have provided some qualitative explanation of our results based on inspection of some classes and of system release histories.

Threats to **external validity** concern the possibility to generalise our results. First, we studied four systems having different size and belonging to different domains. Yet, further validation on a larger set of systems should be performed. Second, we used a representative subset of APs and DPs. Different APs and DPs could have led to different results and should be studied in future work.

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 and CVS/SVN repositories of the studied systems are available to obtain the same data. The raw data used to compute the statistics is available on-line[2].

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention to the assumptions of the statistical tests. Also, we used non-parametric tests that do not require the normality of the distribution of the data. One possible threat concerns **RQ3**, in which we only analysed interactions between APs and DPs, for which APs alone had a significant effect on class change-proneness. We could have miss some significant interactions. Indeed, there could be significant interactions between some APs and DPs while the unique participation of a class in these APs and the DPs does not have any statistically significant effect. However, the available data set does not allow to analyse all possible combinations of APs and DPs without risk of Type II error.

7.6 Summary

In this chapter, we investigated (1) the co-occurrence, in ArgoUML, Eclipse-JDT, Mylyn, and Rhino, of antipatterns and design patterns and (2) the relations of co-occurrences with class change-proneness.

Study results showed that the percentages of classes that participate in co-occurrences of antipatterns and design patterns range between 28% and 68%. They also showed that classes participating to antipatterns, design patterns, and both increase/decrease consistently with the systems sizes. In some system, for example ArgoUML, the numbers of occurrences of antipatterns tend to increase in some releases—indicating possible code decay—while in other releases they decrease while that of design patterns increase, hint of code improvements/re-structuring activities.

Study results also showed that, in all system but Eclipse-JDT, class change-proneness odds ratios significantly decrease for classes participating in both antipatterns and design patterns *w.r.t.* classes participating in antipatterns only. Thus, we empirically found a

tangible, positive effect of design patterns: when a class is properly designed using some design patterns, even if it participate in (or decays towards) antipatterns, the negative effect of the antipatterns is mitigated by the robustness from the design patterns.

This effect of interactions between APs and DPs should be considered carefully when building models combining design patterns and antipatterns because it can affect the accuracy of their classification. In Chapter 8, we choose Bayesian Belief Networks because they are robust and their classification is not affected by dependencies among variables [Zhang, 2004].

Chapter 8

Design-based Quality Models

If you can not measure it, you can not improve it.
Lord Kelvin (1824-1907).

Quality models are increasingly important because of the growing complexity and pervasiveness of software systems. Moreover, the current trend in outsourcing development and maintenance requires means to measure quality with great details. Change- and fault-prone classes in systems increase system costs by requiring more efforts and time from developers and maintainers. Therefore, identifying and characterizing change- and fault-prone classes can enable developers and maintainers to focus timely preventive actions, such as peer-reviews, testing, inspections, and restructuring efforts on the classes with the similar characteristics in the future [Koru and Liu, 2007].

Many quality models in the literature have been proposed to assess the change- and fault-proneness of classes. Yet, as remarked by Briand and Wust [2002] in their survey of quality models, none of them considered the organisation of classes. They are all limited to internal attribute of classes among which size, cyclomatic complexity, and coupling.

In Chapter 5 and Chapter 6 we showed that design motifs and antipatterns impact significantly the change- and fault-proneness of classes. Using these results, and following our method DEQUALITE, we present in this chapter two quality models to predict change- and fault-prone classes. Relating change- and fault-proneness to design motifs and antipatterns rather than to metrics has the major advantage that the resulting quality models can tell the developers whether a design choice is “poor” or not.

8.1 Goal

Our goal in this chapter is to build prediction models to help developers determine where to focus their inspection efforts in systems. We are particularly interested in predicting change- and fault-prone classes that can then be targeted by specific reviews, verification, and maintenance activities. Though many studies have been reported on detecting change- and fault-prone classes in object-oriented systems, *eg.*, [Briand *et al.*, 2002 ; Koru and Liu, 2007], the specificity of our approach in this dissertation is our focus on the design of systems. We believe that not only the internal structure of classes but also their organisation is important to effectively assess their quality. Chapters 5 and 6 have showed that playing roles in design motifs or being involved in antipatterns have a significant impact on the change- and fault-proneness of classes.

We follow our method DEQUALITE and use these results to build Bayesian Belief Networks (BBNs) for the prediction of change- and fault-prone classes in systems. We aim at achieving high precision and recall. We chose BBNs because, contrary to other machine learning techniques, they are robust in the sense that they can also work with missing data and allow quality analysts to specify explicitly their decision process. When data is unavailable or must be adapted to a different context, an analyst can encode her judgement into a BBN. BBNs have been successfully used in fields as diverse as medicine [Szolovitz, 1995], risk management [Cowell *et al.*, 2007], and computer science [Fenton and Neil, 2007].

8.2 Building BBNs Quality Models

Following our method DEQUALITE presented in Chapter 1, we have selected the change- and fault-proneness quality attributes (Step 1). We have chosen the design specifications: design patterns, and antipatterns; and have assessed their impact of systems' quality in Chapters 5 and 6 (Step 2).

In this section, we propose and build a BBN to relate the design of systems to quality attributes, *i.e.*, class designs (playing roles in design patterns or participating to antipatterns) to their change- (respectively fault-) proneness, as specify by Step 3.

The structure of this BBN is as follows: the input nodes correspond to the characterizations of the design of a class and the output node is the probability that the class is change- (respectively fault-) prone.

A BBN is a directed, acyclic graph that represents a probability distribution [Pearl, 1988]. In this graph, each random variable X_i is denoted by a node. A directed edge between two nodes indicates a probabilistic dependency from the variable denoted by the parent node to that of the child. Therefore, the structure of the network denotes the assumption that the values of each node X_i in the network are only conditionally dependent on its parents. Each node X_i in the network is associated with a conditional-probability table that specifies the probability distribution of all of its possible values, for every possible combination of values of its parent nodes.

Formally, a BBN is a classification function (see Section 4.6) $f\colon R^d \mapsto \mathrm{C}$ that assign a label from a finite set of classes $\mathrm{C} = \{c_1, \ldots, c_q\}$ to observations $\mathbf{x} = (a_1, \ldots, a_d) \in R^d$.

In the case of change- (respectively fault-) proneness, there are two possible outputs for the classification of a given object-oriented class:
$\mathrm{C} = \{change - prone, not\ change - prone\}$ (respectively $\mathrm{C} = \{fault - prone, not\ fault - prone\}$), given an observation $(a_1, ..., a_d)$ *i.e.*, a vector of inputs describing the design of the class in terms of metrics, or number of antipatterns and–or design motifs (*eg.*, a_i can be AP, stating that the class participates in an antipattern; with AP taking the values no role, one role, and more role, when a class participates to zero, one and many antipatterns).

A quality analyst needs two pieces of information to build such a BBN: the structure of the network, in the form of nodes and arcs (causal relations) and the conditional-probability tables describing the decision processes between each node. By structuring the network, the quality analyst ensures that the decision process is valid. The conditional probabilities can be learned using historical data or entered directly by the quality analysts when data is missing. The structure ensures the qualitative validity of the approach while appropriate conditional tables ensure that the BBN is well-calibrated and is quantitatively valid. BBNs are robust and their classification is not affected by dependencies among nodes [Zhang, 2004]. Therefore, enabling us to handle co-occurrences of antipatterns and design motifs. More details on Bayesian classifiers are presented in Section 4.6.

Figure 8.1 presents the structure of a BBN for change proneness. We use it as our running example and explain the structure of our BBNs quality models.

Input Nodes. The input nodes of the BBN from Figure 8.1 are AP and DP. The node AP captures the structure of a class in term of antipattern. We use DECOR presented in Chapter 4 to specify and detect antipatterns in classes. A class can participates in zero, one, or more antipatterns. We discretise this node into three different levels: no smell,

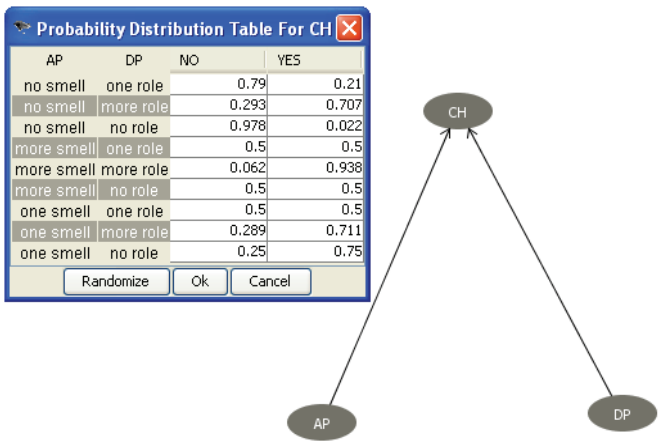


Figure 8.1 – Bayesian Belief Network quality model for change-proneness.

one smell, and more smell. The node DP captures the structure of a class in term of design motifs. We use DeMIMA presented in Chapter 4 to specify and detect design motifs in classes. Similarly to antipatterns, a class can play zero, one, or more roles in a design motif. We discretise this node into the three levels: no role, one role, and more role.

For each node, we compute the probability distributions using historical data as follows:

- For AP, we use the frequencies of classes participating respectively in no smell, one smell, and more smell to compute the probability that a given class c_i participates respectively in no smell, one smell, and more smell .
- Similarly, again using the frequencies of classes playing respectively no role, one role, and more role in a design motif, we compute for DP the probability that a given class c_i plays respectively no role, one role, and more role in a design motif.

Output Nodes. The probability of the output node *i.e.*, change-proneness is inferred from the probabilities of input nodes (AP and DP) using Bayes' theorem. Every output node has a conditional probability table to describe the decision given a set of inputs. For our change-proneness BBN from Figure 8.1, the probability of a class to change depends directly on the two symptoms: AP and DP.

We can use previously tagged data to fill the conditional probability table with: $P(CH|one\ smell; no\ role), P(CH|one\ smell; one\ role), P(CH|one\ smell; more\ role)$,

$P(CH|no\ smell; no\ role), P(CH|no\ smell; one\ role), P(CH|no\ smell; more\ role),$
$P(CH|more\ smell; no\ role), P(CH|more\ smell; one\ role), P(CH|more\ smell; more\ role).$

8.3 Validation and Refinement

Following Step 4 of DEQUALITE, we now perform some experiments to validate and refine the BBN built in Section 8.2. We aim at answering the two following research questions:

RQ1: *To what extent a BBN quality model built using our method is able to predict change/fault-prone classes in a system?*

RQ2: *Are the results of a BBN built using our method better than state-of-the-art prediction models with metrics?*

8.3.1 Experimental Design

The *goal* of these experiments is to improve the quality of systems by predicting change- and fault-prone classes. Our *purpose* is to provide a quality model that takes into account the design of systems. The *quality focus* is to provide a sorted set of change- or fault-prone classes that prioritise the most probable change- or fault-prone classes. The *perspective* is that of quality analysts, who perform evaluation activities and are interested in locating parts of a system that need improvements with the least possible efforts. The *context* of this experiment is both development and maintenance.

We use the following three open-source Java systems: Rhino, Mylyn, and EclipseJDT. They have been used in previous Chapters 6 and 7 and are described in Section 4.5.

8.3.2 Analysis Method

To answer RQ1, we studied the accuracy of our BBNs in two scenarios. First, we assumed that there is historical data available for a given system (*i.e.*, data on changes or faults). This data was used to calibrate a BBN, which was then applied on the same system and the predicted classes compared with the expected classes. Second, we studied the accuracy of our BBNs using heterogeneous data: we calibrated the BBNs using change-prone classes from one system and applied them on another system.

To answer RQ2, we replicated a study by Zimmermann *et al.* [2007] and showed that models built from design data *i.e.*, AP and DP, produce equivalent or better results than models built from metrics only.

8.3.2.1 Scenario 1: Intra-system Validation

In this first scenario, we studied how the change history of a system can be used to predict change-prone classes in future releases of the system. We train the model on the first release of a system and use it to predict change-prone classes in future releases. We chose to train on the first release because we wanted to capture the impact of the design of newly introduced classes on their change-proneness. For each class of the system, the BBN returned a probability that the class will change and also its probability to remain stable, *i.e.*, not changed. We ranked the classes according to these probabilities. Figure 8.2 presents the result achieved on Rhino where the model is trained on release 1.4R3 and tested on release 1.5R1. Each sub-figure shows average precision/recall curves on future releases.

Generally, we are able to achieve a precision above 80% in predicting stable, and change-prone classes on all future releases of Rhino. Moreover, Figure 8.2 shows that our model achieved in average 100 % precision on the 33 first ranked classes (top 28% of candidates reported to be change-prone).

In addition to predicting change-prone classes, our BBN quality model from Figure 8.2 offer for each class, the list of antipatterns and design motifs on the class, obtained with DECOR and DeMIMA. A quality analyst can therefore focus her effort on restructuring the top change-prone classes to improve the overall quality of the system.

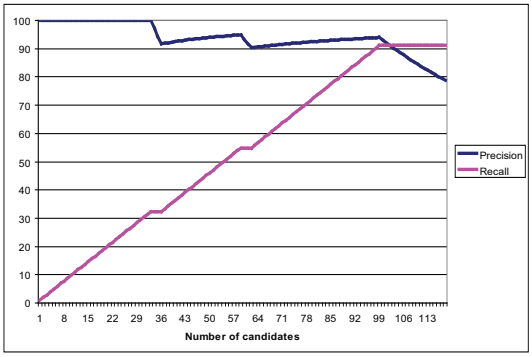
We replicate these results on Mylyn for the releases 2.0.0 and 2.1, and obtained 100% precision on the top 95 first classes (top 29% of candidates reported to be change-prone). Confirming the effectiveness of our BBN quality model in predicting change-prone classes.

Having obtained high precision for this intra-system scenario, we want to understand if BBNs built from DEQUALITE to predict change-prone classes could be effective if applied across systems.

8.3.2.2 Scenario 2: Inter-system Validation

In this second scenario, we assumed that a quality analyst has access to historical data from another system. She would therefore calibrate the BBNs using this data and apply the BBNs on her other system.

(a) Change: Rhino



(b) Stable: Rhino

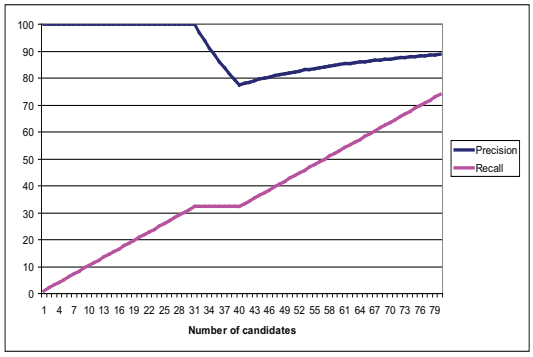


Figure 8.2 – Intra-system calibration: precision and recall.

As shown in Figure 8.3, we can see that with the change history of Rhino, our BBN is able to achieve a good precision on predicting change-prone classes in Mylyn (above 50%) and an excellent precision in predicting stable classes (above 90%). Reciprocally, the BBN trained on Mylyn achieved precisions above 90% in predicting change-prone classes in Rhino and precision above 60% for stable classes.

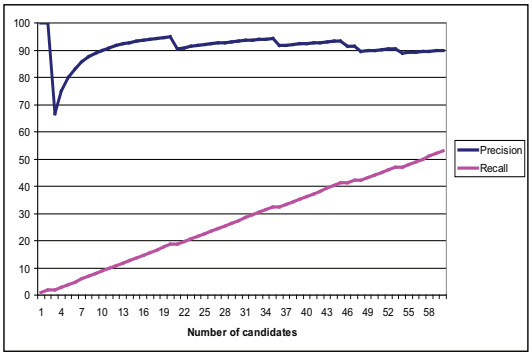
These results suggest that even in the absence of historical data on a specific system, a quality analyst can use a BBN calibrated on different systems and obtain acceptable precision and recall. These results also show that a BBN could be built using data external to a company and then be adapted and applied in this company successfully.

8.3.2.3 Comparison with State–of–the–art Metric Models

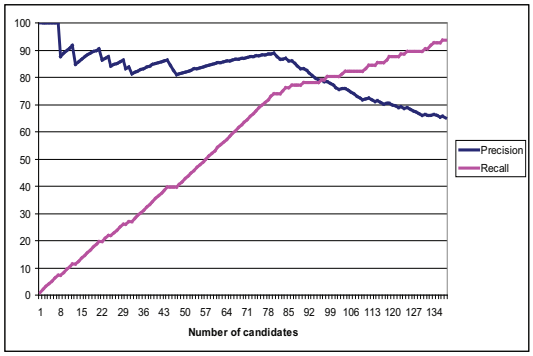
We now replicate a study by Zimmermann *et al.* [2007] and compare the predictive powers of their structural metrics to those of our design data on antipatterns (AP) and design motifs (DP).

Zimmermann *et al.* [2007] mapped faults from the issues reporting system of Eclipse to source code locations for the releases, 2.0 and 2.1, and reported results of their experiment on building prediction models. For each class of the systems, they proposed a list of metrics values, the number of pre-release faults that were reported in the last six months before release and the number of post-release faults that were reported in the first six months after release. They computed Spearman's correlation between the number of pre-release and post-release faults and the complexity metrics in their data set. Between the measure

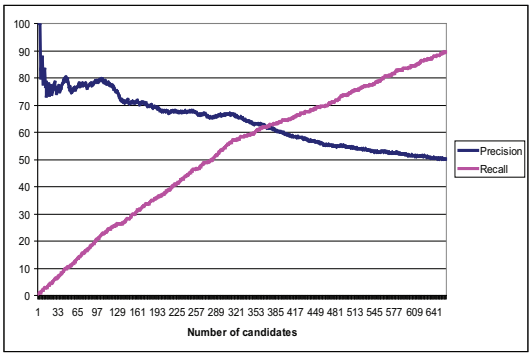
(a) Change – Rhino (calibration: Mylyn)



(b) Stable – Rhino (calibration: Mylyn)



(c) Change – Mylyn (calibration: Rhino)



(d) Stable – Mylyn (calibration: Rhino)

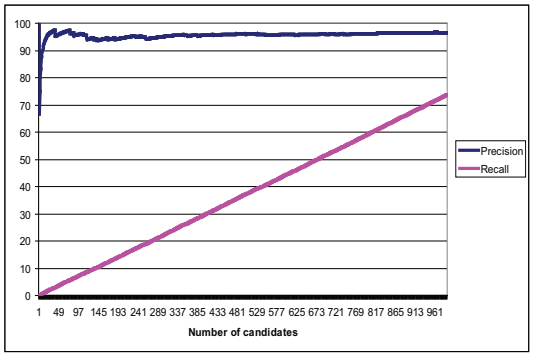


Figure 8.3 – Inter-system validation.

of pre-release faults (*pre*) and post release faults (*post*), they obtained a high correlation of 0.91, but for the complexity metrics, they found only the following to be significant with correlation values above 0.40; accumulated McCabe complexity (VG_{sum}), the total lines of code ($TLOC$), and the closely related sum of method lines of code ($MLOC_{sum}$).

In this section, we reuse the data from their study, which they have made publicly available and replicate their logistic regression models. Specifically, we build three groups of logistic model:

Metrics: in this group, we replicate the logistic regression models built from the four metrics: *pre*, VG_{sum}, $TLOC$, and $MLOC_{sum}$ [Zimmermann *et al.*, 2007].

Design: this group concerns models built using only design data, *i.e.*, number of roles played in design motifs (DP) and number of antipatterns in classes (AP).

Metrics + Design: models in this group are built on a combination of metrics and design data from the two previous groups *i.e.*, *pre*, VG_{sum}, $TLOC$, $MLOC_{sum}$, DP, and AP.

Table 8.1 presents the coefficients of the logistic regression with metrics and design data, and Table 8.2 reports the results achieve by the models from the three groups. Consistently, models from the group *Metrics + Design* (highlighted in bold face) always outperform models built with metrics only and the contribution of design data in this model is superior to the contribution of metrics.

Table 8.1 – Coefficients of the Logistic regression with metrics and design data.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-3.4222	0.3160	-10.83	**0.0000**
pre	0.2021	0.0325	6.23	**0.0000**
MLOC_sum	0.0087	0.0032	2.72	**0.0065**
TLOC	-0.0042	0.0026	-1.59	0.1120
VG_sum	0.0025	0.0053	0.47	0.6366
AP_one smell	0.8158	0.3466	2.35	**0.0186**
AP_more smell	1.5209	0.2872	5.29	**0.0000**
DP_one role	-0.3534	0.3331	-1.06	0.2887
DP_more role	0.4966	0.1705	2.91	**0.0036**

This result confirms the intuition behind our work and our thesis that the design of systems is important to effectively assess their quality.

A model taking into account the design of system have a better accuracy in predicting fault-prone classes than a model based on metrics solely.

We performed a stepwise logistic regression to confirm this result. The stepwise logistic regression screens an available list of independent variables to select only those that it deems *important* in describing the dependent variable [Sheskin, 2007]. Among the six variables: *pre*, VG_{sum}, $TLOC$, $MLOC_{sum}$, DP, and AP, the stepwise logistic regression retained *pre*, $MLOC_{sum}$, $TLOC$, AP, and DP confirming the importance of design data (AP, and DP) to predict fault-prone classes. Consequently, we conclude that design data are orthogonal to metrics. They provide a complementary information on the organisation of classes and appear to be important for a good prediction of fault-prone classes. Moreover, as presented on Table 8.2, design data can be in some context more important

Table 8.2 – Logistic regression: Precision and Recall.

Training	Testing	Metrics		Design		Metrics+Design	
		Precision	Recall	Precision	Recall	Precision	Recall
2.0	2.0	0.68	0.22	0.63	0.12	**0.71**	0.24
	2.1	0.42	0.25	**0.65**	0.13	**0.44**	0.26
2.1	2.1	0.61	0.16	**0.64**	0.14	**0.62**	0.17
	2.0	0.60	0.11	0.58	0.13	**0.62**	0.12

than metrics; on Eclipse JDT 2.1, models built with design data only outperform models from the two other groups.

8.3.3 Refinement of our BBN Model

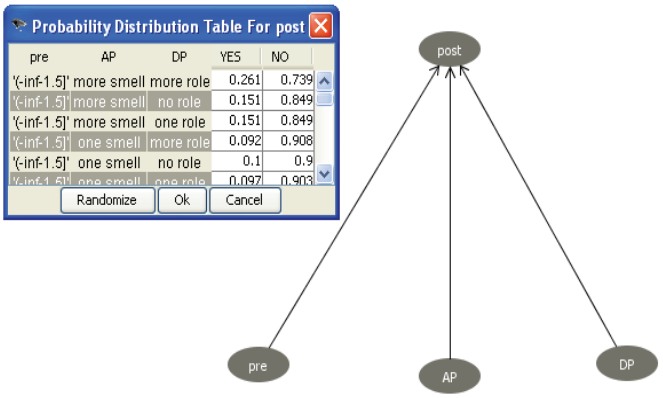
Following the results of our stepwise regression, we improve our design-based BBN model with information from the metrics *pre*, $MLOC_{sum}$, and $TLOC$. Figure 8.3(b) presents the structure of the resulting model. We trained and apply this improved model on releases 2.0 and 2.1 of Eclipse JDT and achieved the performance summarizes in Table 8.3.

Table 8.3 – Bayesian Belief Networks: Precision and Recall.

Training	Testing	Design		Design + selected Metrics	
		Precision	Recall	Precision	Recall
2.0	2.0	0.61	0.13	0.63	0.27
	2.1	**0.65**	0.15	0.41	**0.31**
2.1	2.1	**0.65**	0.14	0.50	**0.24**
	2.0	0.54	0.18	0.62	0.22

In general, the BBN improved with metrics outperforms the BBN built on design data solely except for release 2.1 of Eclipse JDT. This result is consistent with those achieved by the logistic regression in Table 8.2. We explain the exception of release 2.1 by the fact that the stepwise logistic regression, being a data-driven method, does not necessarily identify "best models" as they work by fitting an automated model to the current dataset, hence raising the danger of overfitting to noise in the dataset at hand [Shtatland *et al.*, 2001]. For the release 2.1 of Eclipse JDT, the best model is the BBN built on design data solely. In Chapter 7, we had already observed the peculiarity of this release.

(a) Design



(b) Combination of Design and Metrics

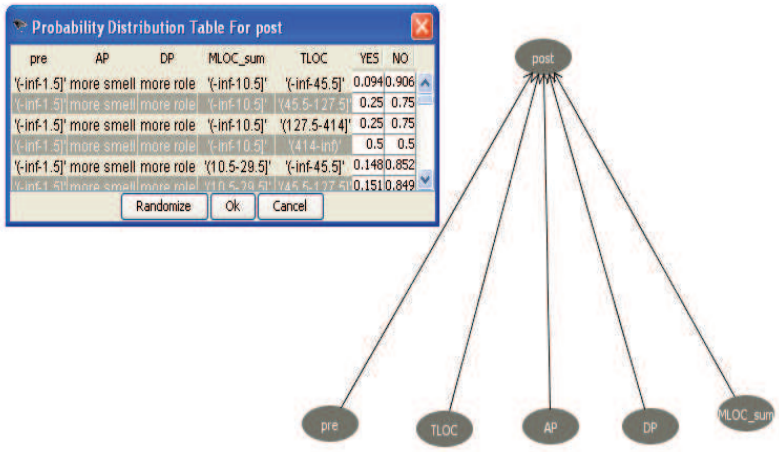


Figure 8.4 – Bayesian Belief Networks for fault-proneness.

8.4 Discussions

We now discuss the experiments and the use of DEQUALITE by quality analysts based on the results of the experiments.

8.4.1 Using BBNs in an Industrial Context

We showed that our BBNs are able to efficiently prioritise candidate change/fault-prone classes that should be inspected by a quality analyst. In Section 8.3.2.2, we showed that BBNs built using DEQUALITE and calibrated using external data, can successfully identify change-prone classes. The systems were of different nature (a JavaScript interpreter vs. an Eclipse plug-in) and developed by different development teams. Therefore, quality analysts in their industrial contexts could use public available data from open source systems and calibrate BBNs to predict change-prone classes on their systems.

Although recall values of our BBNs models of fault-proneness are relatively low, the precisions are generally above 60%, suggesting that classes classified as fault-prone by our BBNs are most likely the case. We attribute the low recall values to the small number of faulty classes in our training data sets; predicting a rare event is difficult in general.

As confirmed by experiments, BBNs models built with DEQUALITE on design data are better tools than models based on metrics solely. Moreover, because they rely on design information like antipatterns and design motifs which refer to specific programming styles, they are able to tell a developer the exact design smell affecting a class, while models based on metrics only provide a score and let the developer identify problems alone.

Moreover, as discussed in Section 8.3.3, a BBN model built with DEQUALITE can be customized and improved with other data that a quality analyst may consider relevant in her industrial context. Stepwise logistic regression provides an interesting tool for model improvements. Indeed, beside the fact that it is a data-driven method, the stepwise logistic regression is a very interesting tool for the exploration of models as it "allows for the examination of a collection of models which might not otherwise have been examined" [Shtatland *et al.*, 2001].

8.4.2 Estimating the Number of Change/Fault-prone Classes in Systems

Estimating the number of occurrences of change/fault-prone classes in a system is important to stop investigating spurious candidate classes. A well-calibrated BBN should be able to estimate this number in a system.

In practice, when quality analysts are confronted with long ranked lists of candidates classes, they employ ad-hoc methods to reduce their efforts while improving their utility:

- They limit the manual validation to the top 20% of candidate classes because Pareto's law suggests that change/fault-prone classes (in general) are located in only 20% of the code.
- They count the distance between every two true positive and stop inspecting candidate classes as soon as this distance is greater than the average distance of all previous two candidates.

We contribute to these effort-reduction methods by proposing a variant of the Rank-Biased Precision (RBP) [Moffat and Zobel, 2008] to compute a stop point at which the utility of a quality analyst is maximal.

The RBP is a metric design to reflects the satisfaction of a quality analyst in absolute terms. RBP assumes that a quality analyst always starts by examining the top-ranked classes of a list of candidates classes, progressing from one class to the next with a probability p, and, conversely, end her examination of the ranking at a point with probability $1 - p$. Each termination is decided independently of the current depth reached in the ranking, of previous decisions, and of whether or not the class just examined was relevant or not.

RBP also assumes that, as a quality analyst skims through candidate classes, they are willing to pay \$1 for each true positive found, but nothing for false positives. This assumption leads to a notion of income for prediction models, based on the utility gained by the quality analysts. As the quality analysts progress down the ranked lists of candidate classes, they are thus running up an account with the prediction model, or, equivalently, increasing their total utility every time they find a true positive. The total expected utility derived by the quality analyst (RBP), and the income payable to the prediction model, are given by:

$$RBP(p) = (1 - p) \sum_{i=1}^{d} r_i p^{(i-1)} \tag{8.1}$$

where $r_i \in \{0, 1\}$ is the relevance judgement of the i^{th} ranked class (which is 1 if the class is a true positive and 0 otherwise), d is the length of the list, and the $(1-p)$ factor is used to scale the RBP within the range $[0, 1]$.

We extend this metric and assign a cost θ for false positives and redefine RBP as follows:

$$RBP(p) = (1 - p) \sum_{i=1}^{d} (r_i \lambda_i + (1 - \lambda_i)(-\theta)) p^{(i-1)} \tag{8.2}$$

where $r_i \in \{0, 1\}$ is the relevance judgement of the i^{th} ranked class, λ_i the probability provided by the BBN that the i^{th} ranked class is a true positive, d the length of the list, and the $(1 - p)$ factor is used to scale the RBP within the range $[0, 1]$.

The optimum of this new definition of RBP is reached as soon as $\lambda_i \leq \frac{\theta}{\theta+1}$ at a rank i.

Therefore, if a quality analyst values the cost of her reviewing a false candidate to -$1, she should stop reviewing candidate classes as soon as the probability of the candidates is under 0.5, because her maximal utility would then be reached.

The value of the RBP metric depends on the precision of the BBN. Therefore, we suggest that quality analysts apply this effort-reduction method more than once. After reaching the stop point, quality analysts should re-calibrate the BBN to improve its precision as well as the estimation of the stop point.

8.4.3 DEQUALITE vs. QMOOD

The hierarchical model QMOOD, introduced by Bansiya and Davis [2002], is the latest major quality model and also the most-used referenced model in recent studies. The success of this model can be explain by the fact that it has been validated on many medium and large industrial systems. The quality model QMOOD defines six equations to assess the following quality attributes: reusability, flexibility, understandability, functionality, extendibility, and effectiveness.

To further validate our approach, we want to check if a BBN produced by DEQUALITE could be used to detect the same problematic classes as QMOOD, although the two models measure different quality attributes. Therefore, we implemented the six equations of QMOOD described in [Bansiya and Davis, 2002], and calibrated the BBN from Figure 8.1 on Rhinov.1.4R3 . We then applied the two quality models on Mylynv.2.0.0 and observed the following:

Among the top 20% of classes considered less reusable, less flexible, and less extendible by QMOOD:

- 71% of them were change-prone classes (measured as in Section 4.4);
- 98% of them were predicted as change-prone by the BBN with;
- 69% of these classes being among the top 20% results of the BBN.

Consequently, even though the BBN was not designed to measure the exact same attributes as QMOOD and was calibrated on a different system (Rhino), the results suggest that it can be almost as effective as QMOOD in detecting problematic classes in systems.

Moreover, following our method DEQUALITE, a quality analyst could calibrate the BBNs to predict the same attributes as QMOOD giving the availability of appropriate training sets, *eg.*, manually-validated instances of less flexible classes.

8.5 Summary

In this chapter, we have applied DEQUALITE, and built BBNs models that allows the measurement of the quality of object-oriented systems by taking into account the internal attributes of the system and also its design. Calibration of BBNs is done automatically using Bayes' theorem. Consequently, BBNs have two main benefits with respect to previous approaches: they work with missing data and can be tuned using quality analysts' knowledge. In addition, candidate classes *i.e.*, potential change/fault-prone classes, are associated with probabilities, which indicate the degree of uncertainty that a class is indeed to be change/fault-prone in future. These probabilities can help focus manual inspection by ranking the candidate classes.

To validate DEQUALITE, we first built BBNs for change-proneness; we calibrated, and evaluated them on two systems, Rhino and Mylyn. The results showed high precision and recall and the capability of the BBNs to assign high probabilities to candidate classes that are *indeed* change-prone. Second, we showed that the results of the BBNs obtained from DEQUALITE are in general equivalent or superior to these of a state-of-the-art model with metrics and that when BBNs are improved with metrics, their accuracy increase. Third, we showed that BBNs obtained from DEQUALITE could be as effective as QMOOD in detecting problematic classes in systems.

We also discussed the applicability and utility of DEQUALITE in an industrial context and proposed a method to estimates the number of problematic classes in a system and thus, reduce quality analysts' efforts. The BBNs presented in this Chapter are available on-line in our portal SQUANER[1]

[1] `http://www.squaner.khomh.net/`

Chapter 9

Conclusions

In this chapter, we summarise the results and conclusions of our dissertation. We also discuss opportunities for extending our work.

9.1 Dissertation Findings and Conclusions

Systems like people get old [Parnas, 1994] their complexity increases overtime and they degrades in effectiveness unless work is done to maintain a good level of quality [Lehman, 1996]. With DEQUALITE, we presented a method to systematically build BBNs-based quality models to predict classes with bad quality in systems. These BBNs-based quality models take into account not just the internal structure of systems (*i.e.*, the structure of their classes), but also their design (*i.e.*, the organisation of their classes). Taking into account the design of systems in quality evaluations is important because this design is the first thing developers should master when they perform changes on a system; a poor design quality often results in *Ignorant surgeries* [Parnas, 1994] that cause systems aging and increase their maintenance costs.

As shown by our experiments, quality models built with DEQUALITE achieve high precision and recall in predicting change-prone classes and results in general equivalent or superior to these of state-of-the-art models with metrics when predicting fault-prone classes. The accuracy of fault-proneness models built with DEQUALITE increases when they are improved with new information on systems, like class sizes.

Moreover, contrary to quality models from the literature, which provide only quality scores, quality models from DEQUALITE, which are BBNs-based, provide in addition to the probability that a class is of bad quality, the list of design smells affecting the

class and also a ranking of all classes with bad quality. A quality analyst can use this ranking to focus timely preventive actions, such as peer-reviews, testing, inspections, and restructuring efforts on classes with high ranks because she knows that such classes have more defects.

These results provide quantitative evidence that design patterns and antipatterns have an impact on the quality of systems and that taking them into account does improve prediction thus proving our thesis.

To apply DEQUALITE, we have performed a series of experiments aimed at understanding and assessing the impact of design patterns and antipatterns on classes change- and fault-proneness. From these experiments, we are able to draw the following conclusions:

1. Counterintuitively, design patterns do not always improve the quality of systems; some design patterns are perceive by developers to decrease some aspects of the quality of systems and to not necessarily promote reusability, expandability, and understandability as claim by [Gamma *et al.*, 1994].

2. Tangled implementations of design patterns exist and significantly affects the structure of classes as well as their change- and fault-proneness. A particular attention should be paid to classes playing roles in design motifs; in particular classes playing two roles, because they have internal and external metric values that are significantly higher than these of other classes: they are more change-prone, less cohesive, more coupled, more complex, and more issue-prone.

3. Classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes than other classes. Developers should pay attention to systems with a high number of them, because these classes are more likely to be the subject of their change efforts. For example, MessageChains which are violations of the Law of Demeter are found consistently related with high fault and change rates.

4. A non negligible percentage of classes participates in co-occurrences of antipatterns and design patterns in systems. On these classes, design patterns have a positive effect in mitigating antipatterns; hinting that when a class is properly designed using some design patterns, even if it participates in (or decays towards) antipatterns, the negative effect of the antipatterns is mitigated by the robustness from the design patterns.

5. IRSs, in open-source development, have a far more complex use than simple bookkeeping of corrective maintenance and studies based on data from these IRSs should carefully consider which issues are used to build their predictive models.

9.2 Opportunities for Future Research

In this dissertation, we have verified our thesis and proved that by considering the presence of design patterns and antipatterns, it is possible to build better quality models than simply by considering internal attributes of classes. This result opens interesting new directions of research including: the evaluation of the quality of multi-language systems considering their design, the usability of quality models, and the extension of DEQUALITE to other sources of information like identifiers and to other quality attributes like understandability.

9.2.1 Quality of Multi-language Systems

The maintenance and evolution of multi-language systems is a neglected topic in the research domain of software evolution. Yet, in large industrial software systems, it is often the case that multiple programming languages are used [Jones, 1998]. Today, more than three programming languages is the rule rather than the exception. For example, J2EE systems use an amalgam of several technologies (*eg.*, Enterprise Java Beans - EJB and Java Server Pages - JSP), each one of them in different languages, (*i.e.*, Java, XML, SQL). Consequently, applying existing reverse engineering and quality assurance techniques developed for mono-language systems fails due to many reasons, including:

- The techniques developed for one language overlook the information written in other languages and their possible mismatch, for example between Java code, XML configuration files, JSP files, and database structure.
- These techniques fail to analyse dependencies existing across languages, such as the request a JSP page may perform to obtain data from a certain database table.
- These techniques lack suitable data models and storage mechanisms to represent and reason about their functioning, reflecting and abstracting their complexity.

The fast-growing number of multi-language systems emphasizes the necessity to overcome the previous limitations and build techniques and tools to help assess their quality.

Therefore, it would be interesting to answer the following research questions:

- How do we model multi-language systems to support analysis and cross-analysis of the different languages?
- How do we define and retrieve design patterns, antipatterns, and code smells in multi-language systems and what are their impacts on the quality of the systems?
- How do design patterns and antipatterns overlap different layers in different programming languages of the multi-language systems?
- How do design patterns and antipatterns evolve in multi-language systems?
- How can we evaluate the quality of multi-language systems by taking into account the impact of their design?

Being able to effectively assess the quality of multi-language systems is very important as it will deepen the understanding of practitioners, developers, quality assurance personnel on their design choices and help them make well informed design and implementation decisions and forecast the impact of these decisions. As a result, a developer may decide to include in a new software system developed using an object-oriented language, an existing predefined and pretested library implemented in a procedural language and thus save cost and effort. Assessing effectively the quality of multi-language systems will improve the reuse and the life of existing software systems, which is often an important reason behind building multi-language software.

9.2.2 Usability of Quality Models

Despite the large number of quality models and publicly available quality assessment tools like PMD, very few studies have investigated the use of quality models by developers in their daily based activities. It would be interesting to understand how these models affect developers behaviors as well as their ability to write good code. I am currently working on the integration of our quality models in Gutsy, a software development monitoring tool built on SVN to automatically provide feedback to developers. With our quality models, Gutsy will provide feedback to developers on the quality of their code each time a new commit is made. With Gutsy, I plan to perform a series of experiments aimed at assessing the usability of a quality model in a software development environment.

9.2.3 Extension of DEQUALITE

Although we have proved in this thesis that antipatterns and design patterns significantly impact systems change- and fault-proneness, it would be interesting to assess their impact on more subjective quality attributes like understandability. I am currently performing a series of controlled experiments to understand the effect of various antipatterns on the understandability of systems. Promising results have already been obtained for the God Class and the Spaghetti Code. With these results, I will be able to build with DEQUALITE, quality models for the prediction of classes understandability.

In the near future, I plan to investigate new sources of information on systems, which could help improve the quality models built with DEQUALITE, such as source code identifiers. Recent studies by [Marcus *et al.*, 2008 ; Butler *et al.*, 2009] have related source code identifiers to quality and showed that the identification of flawed identifiers in systems could help predict faults. It would be interesting to investigate whether a combination of system design and the quality of its identifiers improves quality assessment.

Related Publications

The following is a list of our publications related to this dissertation.

Articles in journal

1. **Foutse Khomh**, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, (2010) An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness, Journal of Empirical Software Engineering (EMSE) (revised and resubmitted).

2. **Foutse Khomh**, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui, (2010) BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns, Journal of Systems and Software, special issue of QSIC 2009 (JSS) (revised and resubmitted).

Book chapters

1. **Foutse Khomh** and Yann-Gaël Guéhéneuc, (2010) Construction de modèles de qualité prenant en compte la conception des systèmes et présentation d'un tel modèle de qualité, Evolution et Rénovation des Systèmes Logiciels , Hermes, (To appear)

Conference articles

1. Salima Hassaine, **Foutse Khomh**, Yann-Gaël Guéhéneuc, and Sylvie Hamel (2010) IDS: An Immunology-inspired Approach for the Detection of Software Design Smells, In Proceedings of the Quality in Reengineering and Refactoring track at the 7th International Conference on the Quality of Information and Communications Technology (QUATIC), September 29 - October 2, 2010, Oporto, Portugal. IEEE Computer Society Press.

2. Rocco Oliveto, **Foutse Khomh**, Giuliano Antoniol, and Yann-Gaël Guéhéneuc (2010) Numerical Signatures of Antipatterns: An Approach based on B-Splines, In Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR), March 15-18, 2010, Madrid, Spain. IEEE Computer Society Press.

3. **Foutse Khomh**, Massimiliano Di Penta and Yann-Gaël Guéhéneuc, (2009) An Exploratory Study of the Impact of Code Smells on Software Change-proneness, In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), October 13-16, Lille, France. IEEE Computer Society Press.

4. Stéphane Vaucher, **Foutse Khomh**, Naouel Moha and Yann-Gaël Guéhéneuc, (2009) Tracking Design Smells: Lessons from a Study of God Classes, In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), October 13-16, Lille, France. IEEE Computer Society Press.

5. **Foutse Khomh**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, (2009) Playing Roles in Design Patterns: An Empirical Descriptive and Analytic Study, In Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM), September 20-26, Edmonton, Alberta, Canada. IEEE Computer Society Press.

6. **Foutse Khomh**, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui, (2009) A Bayesian Approach for the Detection of Code and Design Smells, In Proceedings of the 9th International Conference on Quality Software (QSIC), August 24-25, Jeju, Korea. IEEE Computer Society Press.

7. **Foutse Khomh**, Yann-Gael Gueheneuc, (2008) Do Design Patterns Impact Software Quality Positively?, In Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), du 1-4 avril, Athènes, Grèce. IEEE Computer Society Press.

8. Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, **Foutse Khomh**, Yann-Gaël Guéhéneuc, (2008) Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests, In Proceedings of the 18th IBM Centers for Advanced Studies Conference (CASCON), Toronto, CA, October 27 - 30. ACM Press.

9. Naouel Moha, **Foutse Khomh**, Yann-Gaël Guéhéneuc, (2008) Génération automatique d'algorithmes de détection des défauts de conception, In Proceedings of the 14ème Colloque International sur les Langages et Modèles à Objet (LMO), du 2 -7 mars, Montréal, Quebec, Canada. Éditions Cépaduès.

10. **Foutse Khomh**, (2009) SQUAD: Software Quality Understanding through the Analysis of Design, Doctoral Symposium, 16th Working Conference on Reverse Engineering (WCRE), October 13-16, Lille, France. IEEE Computer Society Press.

11. **Foutse Khomh**, Yann-Gaël Guéhéneuc, (2008) DEQUALITE: Building Design-based Software Quality Models, In Proceedings of the 2nd PLoP Workshop on Software Patterns and Quality (SPAQu), October 18-20, Nashville, Tennessee, USA. ACM Press.

12. **Foutse Khomh**, Yann-Gael Gueheneuc, (2007) Perception and Reality: What are Design Patterns Good For? In Proceedings of the 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), July 31st, Berlin, Germany. Springer-Verlag.

Posters and tools

1. Nicolas Haderer, **Foutse Khomh**, and Giuliano Antoniol, (2010) SQUANER: A Framework for Monitoring the Quality of Software Systems, In Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM), Tool Demonstrations track, September 12-18, 2010, Timisoara, Romania. IEEE Computer Society Press.

2. **Foutse Khomh**, (2009) SQUAD: Software Quality Understanding through the Analysis of Design, Consortium for Software Engineering Research (CSER), April 26-27, Montréal, Canada.

3. Yann-Gaël Guéhéneuc, Janice Ka-Yee Ng, Duc-Loc Huynh, **Foutse Khomh**, (2006) Ptidej: A Tool Suite, IBM CASCON, Oct, 2006, Toronto, Canada.

Technical reports

1. **Foutse Khomh**, Massimiliano Di Penta and Yann-Gaël Guéhéneuc, (2009) An Exploratory Study of the Impact of Code Smells on Software Change-proneness,Technical report, Ecole Polytechnique de Montréal.

2. **Foutse Khomh**, Massimiliano Di Penta, Yann-Gaël Guéhéneuc and Guiliano Antoniol, (2009) An Exploratory Study of the Impact of Antipatterns on Software Changeability, Technical report EPM-RT-2009-02, Ecole Polytechnique de Montréal.

3. **Foutse Khomh**, Yann-Gaäel Guéhéneuc and Giuliano Antoniol, (2009) Playing Roles in Design Patterns: An Empirical Descriptive and Analytic Study, Technical report EPM-RT-2009-03, Ecole Polytechnique de Montréal.

4. **Foutse Khomh**, Naouel Moha and Yann-Gaël Guéhéneuc, (2009) DEQUALITE : méthode de construction de modèles de qualité prenant en compte la conception des systèmes, Technical report EPM-RT-2009-04, Ecole Polytechnique de Montréal.

5. Simon Denier, **Foutse Khomh**, and Yann-Gael Guéhéneuc, (2008) Reverse-Engineering the Literature on Design Patterns and Reverse-Engineering, Technical report EPM-RT-2008-09, Ecole Polytechnique de Montréal.

6. **Foutse Khomh** and Yann-Gael Guéhéneuc, (2008) An Empirical Study of Design Patterns and Software Quality, Technical report 1315, University of Montréal.

Bibliography

[Antoniol *et al.*, 1998] cited page 43
Giuliano Antoniol, Roberto Fiutem, and Lucas Cristoforetti. Design pattern recovery in object-oriented software. In Scott Tilley and Giuseppe Visaggio, editors, *Proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[Antoniol *et al.*, 2001] cited page 43
Giuliano Antoniol, Gerardo Casazza, Massimiliano di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59:181–196. Elsevier, November 2001.

[Antoniol *et al.*, 2008] cited pages 23, 97
Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In Mark Vigder and Marsha Chechik, editors, *Proceedings of the* 18^{th} *IBM Centers for Advanced Studies Conference (CASCON)*. ACM Press, October 2008. 15 pages.

[Aplaydin, 2004] cited page 52
Ethem Aplaydin. *Introduction to Machine Learning*. MIT Press, 2004.

[Aversano *et al.*, 2007] cited pages 3, 18, 63, 94, 97, 99, 104, 115
Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *Proc. of the the* 6^{th} *European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, pages 385–394. ACM Press, 2007.

[Ayari *et al.*, 2007] cited page 23
Kamel Ayari, Peyman Meshkinfam, Giuliano Antoniol, and Massimiliano di Penta. Threats on building models from CVS and bugzilla repositories: the mozilla case study. In Bruce Spencer and Margaret-Anne Storey, editors, *Proceedings of the* 17^{th} *IBM Centers for Advanced Studies Conference*. ACM Press, October 2007.

[Bansiya and Davis, 2002] cited pages 2, 8, 17, 23, 134
Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. In IEEE CS Press, editor, *IEEE Trans. on Software Engineering*, 28:4–17, Jan. 2002.

[Basili and Weiss, 1984] cited page 58
R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738. IEEE Computer Society Press, November 1984.

[Beck and Johnson, 1994] cited pages 2, 56
Kent Beck and Ralph E. Johnson. Patterns generate architectures. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of* 8^{th} *European Conference for Object-Oriented Programming*, pages 139–149. Springer-Verlag, July 1994.

[Bianchi *et al.*, 2002] cited page 2
Alessandro Bianchi, Danilo Caivano, and Giuseppe Visaggio. Quality models reuse: Experimentation on field. In *Proc. of the* 26^{th} *Annual International Computer Software and Applications Conf.*, pages 535– 540. IEEE CS Press, 2002.

[Bieman *et al.*, 2001a] cited page 17
James M. Bieman, Roger Alexander, P. Willard Munger III, and Erin Meunier. Software design quality: Style and substance. In *Proc. of the* 4^{th} *Workshop on Software Quality*. ACM Press, Mar. 2001.

[Bieman *et al.*, 2001b] cited pages 57, 74
James M. Bieman, Dolly Jain, and Helen J. Yang. OO design patterns, design structure, and program changes: An industrial case study. In Paolo Nesi, editor, *Proc. of the Int. Conf. on Software Maintenance*, pages 580–589, http://www.dsi.unifi.it/icsm2001/, Nov. 2001. IEEE CS Press.

[Bieman *et al.*, 2003] cited page 17
James Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In Michael Berry and Warren Harrison, editors, *Proceedings of the* 9^{th} *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press, September 2003.

[Bird *et al.*, 2009] cited pages 23, 40
Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, August 2009.

[Boehm *et al.*, 1976] cited page 15
B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592 – 605. IEEE Computer Society Press, 1976.

[Boehm *et al.*, 1978] cited pages 2, 15
B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. *Char-*

acteristics of Software Quality. Elsevier Science Ltd, 1st edition, june 1978. ISBN: 978-0444851055.

[Bois *et al.*, 2006] cited pages 3, 9, 20, 80
Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *Proceedings of the IASTED International Conference on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.

[Briand and Wüst, 2002] cited pages 2, 8, 10, 13, 21, 121
Lionel C. Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. In Marvin Zelkowitz, editor, *Advances in Computers*. Academic Press, 2002.

[Briand *et al.*, 1997] cited page 47
Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In W. Richards Adrion, editor, *Proceedings of the* 19^{th} *International Conference on Software Engineering*, pages 412–421. ACM Press, May 1997.

[Briand *et al.*, 2002] cited pages 11, 121
Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. In IEEE Computer Society, editor, *IEEE Transactions on Software Engineering*, 28(7):706– 720, july 2002.

[Brito e Abreu and Melo, 1996] cited pages 12, 21
Fernando Brito e Abreu and Walcélio Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3 rd International Software Metrics Symposium*, pages 90–99. IEEE Computer Society, March 1996.

[Brito e Abreu *et al.*, 1995] cited page 12
Fernando Brito e Abreu, Miguel Goulão, and Rita Esteves. Toward the design quality evaluation of object-oriented software systems. In Proceedings of the 5th International Conference on Software, editor, *Proceedings of the 5th International Conference on Software Quality*, October 1995.

[Brown *et al.*, 1998] cited pages 3, 20, 22, 28, 29, 45, 46, 80, 85
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, 1^{st} edition, March 1998. ISBN: 0-471-19713-0.

[Brown, 1996] cited page 42
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.

[Bruegge and Dutoit, 1999] cited page 1
Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice Hall, 1st edition, 1999. ISBN: 0-13-489725-0.

[Butler *et al.*, 2009] cited page 140
Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2009.

[Chidamber and Kemerer, 1991] cited page 10
Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 197–211, 1991.

[Chidamber and Kemerer, 1993] cited page 47
Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. Technical report E53-315, MIT Sloan School of Management, December 1993.

[Cohen, 1988] cited pages 51, 62, 95
Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, New York, 2nd edition, 1988.

[Conte and Campbell, 1989] cited page 86
S. D. Conte and R. L. Campbell. A methodology for early software size estimation. Technical report SERC-TR-33-P, Purdue University, jan 1989.

[Coplien and Harrison, 2005] cited page 3
James O. Coplien and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Upper Saddle River, NJ (2005), 1^{st} edition, 2005.

[Coplien, 1991] cited page 28
James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1^{st} edition, August 1991. ISBN: 0-201-54855-0.

[Cowell *et al.*, 2007] cited page 122
R. G. Cowell, R. J. Verrall, and Y. K. Yoon. Modeling operational risk with bayesian networks. In Patrick L. Brockett and Richard D MacMinn, editors, *Journal of Risk and Insurance*, 74(4):795–827, december 2007.

[Daly *et al.*, 1996] cited pages 13, 21
John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132. Springer Netherlands, January 1996.

[Dhambri *et al.*, 2008] cited page 47
Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 279–283. IEEE CS Press, April 2008.

[Di Penta *et al.*, 2008] cited pages 3, 18, 57, 63, 74, 104
Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol.

An empirical study of the relationships between design pattern roles and class change proneness. In Hong Mei and Kenny Wong, editors, *Proceedings of the* 24^{th} *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, September–October 2008. 10 pages.

[Dromey, 1995] cited pages 4, 16, 17
R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146 – 162. IEEE, february 1995.

[Dromey, 1996] cited pages 2, 16, 17
R. Geoff Dromey. Cornering the chimera. In IEEE Software, editor, *IEEE Software*, 13(1):33–43. IEEE, Jan. 1996.

[Duda and Hart, 1973] cited page 54
R. Duda and P. Hart. *Pattern classification and scene analysis*. John Wiley and Sons, 1973.

[Eaddy *et al.*, 2008] cited pages 40, 41, 48, 81, 101, 102
Marc Eaddy, Thomas Zimmermann, Kaitlin 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. IEEE Computer Society Press, July-August 2008.

[Eden and Kazman, 2003] cited pages 2, 22
Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In Laurie Dillon and Walter Tichy, editors, *Proceedings of the* 25^{th} *International Conference on Software Engineering*, pages 149–159. ACM Press, May 2003.

[Emam *et al.*, 2001] cited pages 84, 100
K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, July 2001.

[Fenton and Neil, 1999] cited page 54
Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering*, 25(5):675–689, 1999.

[Fenton and Neil, 2007] cited page 122
Norman Fenton and Martin Neil. Managing risk in the modern world—applications of bayesian networks. Technical report, London Mathematical Society, November 2007.

[Fischer *et al.*, 2003] cited page 48
Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam Netherlands, September 2003. IEEE CS Press.

[Fowler, 1999] cited pages 3, 22, 28, 29, 45, 46, 80
Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999. ISBN: 0-201-48567-2.

[Gamma *et al.*, 1994] cited pages 2, 9, 17, 18, 22, 23, 25, 26, 28, 40, 42, 44, 56, 78, 80, 137
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[Guéhéneuc and Antoniol, 2008] cited pages 2, 17, 42, 44, 45, 48, 56, 104
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 78
Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering (WCRE)*, pages 172–181. IEEE Computer Society Press, November 2004. 10 pages.

[Guéhéneuc *et al.*, 2005] cited page 23
Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Khashayar Khosravi, and Houari Sahraoui. Design patterns as laws of quality. Technical report, University of Montréal, 2005.

[Guéhéneuc, 2005] cited page 44
Yann-Gaël Guéhéneuc. PTIDEJ: Promoting patterns with patterns. In Mohamed E. Fayad, editor, *Proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005. 9 pages.

[Gyimóthy *et al.*, 2005] cited pages 23, 32
Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object oriented metrics on open source software for fault prediction. In Mark Harman, Bogdan Korel, and Panagioties K. Linos, editors, *Transactions on Software Engineering*, 31(10):897–910. IEEE Computer Society Press, October 2005.

[Hannemann and Kiczales, 2002] cited pages 2, 18, 56, 75
Jan Hannemann and Gregor Kiczales. Design pattern implementation in JAVA and AspectJ. In Satoshi Matsuoka, editor, *Proceedings of the* 17^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, November 2002.

[Harrison *et al.*, 1998] cited pages 10, 21
R. Harrison, S. Counsell, and R. Nithi. Coupling metrics for object-oriented design.

In *Proceedings of the Fifth International Software Metrics Symposium*, pages 150–157. IEEE Computer Society, November 1998.

[Harrison *et al.*, 2000] cited pages 13, 14
R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. In Elsevier Science Inc, editor, *Journal of Systems and Software*, 52(2-3):173 – 179. Elsevier Science Inc, June 2000.

[Hassan, 2009] cited page 2
Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[Henderson-Sellers, 1995] cited page 47
Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1^{st} edition, December 1995. ISBN: 01-32-398-729.

[Hitz and Montazeri, 1995] cited page 47
Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the 3^{rd} International Symposium on Applied Corporate Computing*, pages 25–27. Texas A & M University, October 1995.

[Hollander and Wolfe, 1999] cited pages 61, 62
Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods*. John Wiley and Sons, inc., 2nd edition, 1999. ISBN: 0-471-19045-4.

[Hosmer and Lemeshow, 2000] cited pages 32, 86
David Hosmer and Stanley Lemeshow. *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[Ignatios *et al.*, 2003] cited page 20
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 page 20
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.

[ISO 9126, 1991] cited pages 2, 8, 16
ISO 9126. *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use*. ISO/IEC, December 1991. ISO/IEC 9126:1991(E).

[John and Langley, 1995] cited page 54
G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[Jones, 1998] cited page 138
Capers Jones. *Estimating Software Costs*. McGraw-Hill, New York, 1998.

[Kaczor *et al.*, 2009] cited page 44
Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Identification of design motifs with pattern matching algorithms. In Claes Wohlin, editor, *Information and Software Technology (IST)*. Elsevier, August 2009. 46 pages.

[Kampffmeyer and Zschaler, 2007] cited page 17
Holger Kampffmeyer and Steffen Zschaler. Finding the pattern you need: The design pattern intent ontology. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the* 10^{th} *International Conference on Model Driven Engineering Languages and Systems*, pages 211–225. Springer, September–October 2007.

[Keller *et al.*, 1999] cited page 43
Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[Kerievsky, 2004] cited page 19
J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 1st edition, Aug. 2004.

[Khomh and Guéhéneuc, 2008a] cited page 23
Foutse Khomh and Yann-Gaël Guéhéneuc. Do design patterns impact software quality positively? In Christos Tjortjis and Andreas Winter, editors, *Proceedings of the* 12^{th} *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, April 2008. Short Paper. 5 pages.

[Khomh and Guéhéneuc, 2008b] cited pages 25, 26
Foutse Khomh and Yann-Gael Guéhéneuc. An empirical study of design patterns and software quality. Technical report 1315, University of Montréal, january 2008.

[Khomh *et al.*, 2009a] cited page 23
Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In Giuliano Antoniol and Andy Zaidman, editors, *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, October 2009. 10 pages.

[Khomh *et al.*, 2009b] cited pages 46, 47
Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In Choi Byoung-ju, editor, *Proceedings of the* 9^{th} *International Conference on Quality Software (QSIC)*. IEEE Computer Society Press, August 2009. 10 pages.

[Koru and Liu, 2007] cited page 121
A. Gunes Koru and Hongfang Liu. Identifying and characterizing change-prone classes

in two large-scale open-source products. In Elsevier, editor, *Journal of Systems and Software*, 80(1). Elsevier, January 2007.

[Krämer and Prechelt, 1996] cited page 42
Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[Lake and Cook, 1992] cited page 13
Al Lake and Curtis R. Cook. A software complexity metric for c++. Technical report 92-60-03, Oregon State University, 1992.

[Lange and Nakamura, 1995] cited pages 18, 68
Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proc. of the* 10^{th} *annual conference on Object-oriented programming systems, languages, and applications*, pages 342 – 357. ACM Press, 1995.

[Lanza and Marinescu, 2006] cited page 46
Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. ISBN: 3-540-24429-8.

[Lehman, 1996] cited pages 1, 136
M M Lehman. Laws of software evolution revisited. In *In European Workshop on Software Process Technology*, pages 108–124, 1996.

[Lorenz and Kidd, 1994] cited page 47
Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1^{st} edition, July 1994. ISBN: 0-131-79292-X.

[Mantyla, 2003] cited page 45
Mika Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study*. Ph.D. thesis, Helsinki University of Technology, 2003.

[Marcus *et al.*, 2008] cited page 140
Adrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[Marinescu, 2004] cited pages 28, 45, 46
Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pages 350–359. IEEE CS Press, 2004.

[Masuda *et al.*, 1999] cited page 18
Gou Masuda, Norihiro Sakamoto, and Kazuo Ushijima. Evaluation and analysis of applying design patterns. In Keijiro Araki, Bob Balzer, Carlo Ghezzi, Takuya Katayama,

Jeff Kramer, David Notkin, and Dewayne Perry, editors, *Proceedings of the* 2^{nd} *International Workshop on the Principles of Software Evolution*. ACM Press, July 1999.

[McCabe and Butler, 1989] cited page 47
Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[McCall *et al.*, 1977] cited page 14
J. A. McCall, P. K. Richards, and G. F Walters. Factors in software quality. In Nat'l Tech. Information Service, editor, *Nat'l Tech. Information Service*, 1, 2 and 3, 1977.

[McCall, 2001] cited page 2
James A. McCall. Quality factors. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, 1-2:958–ff. John Wiley and Sons, December 2001.

[McNatt and Bieman, 2001] cited pages 3, 9, 17, 56, 75
William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benefits. In T.H. Tse, editor, *Proc. of the* 25^{th} *Computer Software and Applications Conf.*, pages 574–579. IEEE CS Press, Oct. 2001.

[Misic and Tesic, 1998] cited page 12
Vojislav B. Misic and Dejan N. Tesic. Estimation of effort and complexity: An object-oriented case study. In Elsevier, editor, *Journal of Systems and Software*, 41:133 –143. Elsevier, 1998.

[Mitchell, 1997] cited page 52
Tom Mitchell. *Machine Learning*. MIT Press, 1997.

[Moffat and Zobel, 2008] cited page 133
Alistair Moffat and Justin Zobel. Rank-biased precision for measurement of retrieval effectiveness. In ACM, editor, *ACM Transactions on Information Systems (TOIS)*, 27(1). ACM, December 2008.

[Moha *et al.*, 2008a] cited page 80
Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Francoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proceedings of the* 11^{th} *International Conference on Fundamental Approaches to Software Engineering*, pages 276–291. Springer-Verlag, 2008.

[Moha *et al.*, 2008b] cited page 80
Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In Raoul Medina and Sergei Obiedkov, editors, *Proceedings of the* 4^{th} *International Conference on Formal Concept Analysis (ICFCA)*. Springer-Verlag, February 2008. 18 pages.

[Moha *et al.*, 2009] cited pages 45, 46, 101
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. 16 pages.

[Moser *et al.*, 2008] cited page 2
R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the International Conference on Software Engineering*, 2008.

[Munro, 2005] cited pages 45, 46
Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In Filippo Lanubile and Carolyn Seaman, editors, *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[Ng *et al.*, 2007] cited page 18
T.H. Ng, S.C. Cheung, W.K. Chan, and Y.T. Yu. Do maintainers utilize deployed design patterns effectively? In *Proceedings of the* 29^{th} *International Conference on Software Engineering (ICSE'07)*, pages 168–177. IEEE Computer Society, 2007.

[Ng *et al.*, 2009] cited page 44
Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Identification of behavioral and creational design motifs through dynamic analysis. In Maria Tortorella and Aniello Cimitile, editors, *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*. Wiley, November 2009. 30 pages.

[Niere *et al.*, 2001] cited page 43
Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Recovering UML diagrams from Java code using patterns. In Jens H. Jahnke and Conor Ryan, editors, *Proceedings of the* 2^{nd} *workshop on Soft Computing Applied to Software Engineering*, pages 89–97. Springer-Verlag, February 2001.

[Niere *et al.*, 2002] cited page 43
Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In Michal Young and Jeff Magee, editors, *Proceedings of the* 24^{th} *International Conference on Software Engineering*, pages 338–348. ACM Press, May 2002.

[Olbrich *et al.*, 2009] cited pages 21, 63
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 47
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* 14^{th} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010.

[O'Regan, 2002] cited page 8
Gerard O'Regan. *A Practical Approach to Software Quality*. Springer, 1^{st} edition, June 2002. ISBN: 03-879-532-13.

[Parnas, 1994] cited pages 1, 136
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.

[Pearl, 1988] cited page 122
Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1 edition, September 1988. ISBN: 978-1558604797.

[Philippow *et al.*, 2005] cited page 43
Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and System Modeling*, 4(1):55–70. Springer-Verlag, February 2005.

[Pressman, 2001] cited pages 1, 25
Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5^{th} edition, November 2001. ISBN: 0-07-249668-1.

[Quinlan, 1993] cited page 52
J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[Rao and Reddy, 2008] cited page 46
A. Ananda Rao and K. Narendar Reddy. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, March 2008.

[Riel, 1996] cited page 45
Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rumelhart *et al.*, 1986] cited page 52
D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Seemann and von Gudenberg, 1998] cited page 43
Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of JAVA software. In Bill Scherlis, editor, *Proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.

[Sheskin, 2007] cited pages 32, 50, 51, 129
D.J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[Shtatland *et al.*, 2001] cited pages 130, 132
E. S. Shtatland, E. M. Cain, and M. B. Barton. The perils of stepwise logistic regression and how to escape them using information criteria and the output delivery system. In *SUGI'26 Proceeding*, Cary, 2001. NC: SAS Institute, Inc.

[Shull *et al.*, 1996] cited page 42
Forrest Shull, Walcélio Melo, and Victor R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical report CS-TR-3597, Computer Science Department, University of Maryland, January 1996.

[Simon *et al.*, 2001] cited page 46
Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, page 30. IEEE CS Press, 2001. ISBN: 0-7695-1028-0.

[Singh and Goel, 2008] cited page 2
Yogesh Singh and Bindu Goel. An integrated model to predict fault proneness using neural networks. In ASQ, editor, *Software metrics, Measurement, and Analytical Methods*, 10(2), 2008.

[Spinellis, 2008] cited page 59
Diomidis Spinellis. A tale of four kernels. In Wilhem Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the* 30^{th} *International Conference on Software Engineering*, pages 381–390. ACM Press, May 2008.

[Szolovitz, 1995] cited page 122
Peter Szolovitz. Uncertainty and decisions in medical informatics. *Methods of Information in Medicine*, 1995.

[Tatsubori and Chiba, 1998] cited page 19
Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In Jean-Charles Fabre and Shigeru Chiba, editors, *Proceedings of the* 1^{st} *OOPSLA workshop on Reflective Programming in C++ and* Java, pages 56–60. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4.

[Tegarden *et al.*, 1995] cited page 47
David P. Tegarden, Steven D. Sheetz, and David E. Monarchi. A software complexity model of object-oriented systems. In Andrew B. Whinston, Robert W. Blanning, Sudha Ram, and Richard Y. Wang, editors, *Decision Support Systems*, 13(3–4):241–262. Elsevier Science Publishers, March 1995.

[Tonella and Antoniol, 1999] cited page 42
Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In Hongji Yang and Lee White, editors, *Proceedings of the* 15^{th} *International Conference on Software Maintenance*, pages 230–229. IEEE Computer Society Press, August–September 1999.

[Travassos *et al.*, 1999] cited page 45
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* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–56. ACM Press, 1999.

[Tsantalis *et al.*, 2006] cited pages 44, 63
Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11). IEEE Computer Society Press, November 2006.

[van Emden and Moonen, 2002] cited page 46
Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, October 2002.

[Vaucher *et al.*, 2009] cited page 104
Stéphane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, October 2009.

[Venners, 2005] cited pages 3, 9, 56
Bill Venners. How to use design patterns – A conversation with Erich Gamma, part I, May 2005. `http://www.artima.com/lejava/articles/gammadp.html`.

[Vicinanza *et al.*, 1991] cited page 86
S.S. Vicinanza, T. Mukhopadhyay, and M.J. Prietula. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research*, 2(4):243–262, dec 1991.

[Vinayagasundaram and Srivatsa, 2007] cited pages 15, 17
B. Vinayagasundaram and S.K. Srivatsa. Software quality in artificial intelligence system. *Information Technology Journal*, 6(6):835–842, 2007.

[Vokac, 2004] cited pages 3, 19, 32
Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, pages 904–917. IEEE Press, Dec. 2004.

[Wake, 2003] cited page 45
William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN: 0321109295.

[Webster, 1995] cited pages 9, 45
Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1^{st} edition, February 1995. ISBN: 1558513973.

[Wei and Raed, 2007] cited pages 21, 29, 80
Li Wei and Shatnawi Raed. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. In Elsevier, editor, *Journal of Systems and Software*, 80(7). Elsevier, 2007.

[Wendorff, 2001] cited pages 3, 9, 19, 56, 74
Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[Weyuker *et al.*, 2008] cited page 11
Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing negative binomial and recursive partitioning models for fault prediction. In *Proceedings of PROMISE'08*. ACM, 2008.

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

[Wood *et al.*, 1999] cited pages 14, 21
M. Wood, J. Daly, J. Miller, and M. Roper. Multi-method research: An empirical investigation of object-oriented technology. *journal of Systems and Software*, 48(1). Elsevier, 1999.

[Wuyts, 1998] cited page 43
Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *Proceedings of the* 26^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

[Wydaeghe *et al.*, 1998] cited page 20
B. Wydaeghe, K. Verschaeve, B. Michiels, B. Van Damme, E. Arckens, and V. Jonckers. Building an OMT-editor using design patterns: An experience report. In *TOOLS (26)*, pages 20–32. IEEE Computer Society, 1998.

[Yin, 2002] cited pages 76, 101, 118
R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[Zhang, 2004] cited pages 120, 123
Harry Zhang. The optimality of naive bayes. In Valerie Barr and Zdravko Markov, editors, *In FLAIRS Conference*, 2004.

[Zhu *et al.*, 2002] cited page 2
Hong Zhu, Yanlong Zhang, Qingning Huo, and Sue Greenwood. Application of hazard analysis to software quality modelling. In *Proc. of the* 26^{th} *Annual International Computer Software and Applications Conf.* IEEE CS Press, 2002.

[Zimmermann *et al.*, 2007] cited pages 29, 125, 127, 128
Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering*, 2007.

[Zuse, 1991] cited page 10
H. Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co, New York, 1991. ISBN: 978-0899256405.

Appendix A

Definitions of Metrics

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

A.1 Definitions of metrics

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 dissertation.

B.1 Detailed Definitions of the code Smells

In this dissertation 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 dissertation 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, 1999].

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

Detailed figures on systems

We present in this Appendix some detailed figures of the systems studied in Chapter 6.

C.1 Detailed Characteristics of the Analysed Systems

Table C.1 – Analysed releases, LOC, classes, number of changes and faults/issues per release. Changes and faults are counted from one release to the next.

ArgoUML					Eclipse					Mylyn					Rhino				
Release	LOC	Classes	Changes	Issues Fixing Changes	Release	LOC	Classes	Changes	Issues Fixing Changes	Release	LOC	Classes	Changes	Fault Fixing Changes	Release	LOC	Classes	Changes	Fault Fixing Changes
0.10.1	128,585	792	6,618	475	1.0	781,480	4,647	21,553	2,208	1.0.1	207,436	1,625	1,904	122	1.4R3	30,748	89	1,643	266
0.12	145,625	866	3,651	221	2.0	1,249,840	6,742	26,378	3,367	2.0M1	143,009	1,087	325	32	1.5R1	45,699	140	1,330	74
0.14	175,982	1,204	2,002	234	2.1.1	1,797,917	8,730	10,397	2,211	2.0M2	153,464	1,144	2,218	209	1.5R2	56,452	175	118	9
0.16	180,245	1,205	9,751	851	2.1.2	1,799,037	8,732	11,534	1,923	2.0M3	158,087	1,183	7,615	297	1.5R3	56,676	177	782	134
0.18.1	214,967	1,309	3,782	337	2.1.3	1,799,702	8,736	15,560	3,137	2.0.0	X	X	1,823	22	1.5R4	60,353	186	233	3
0.20	279,864	1,619	3,696	418	3.0	2,260,165	11,166	11,582	798	2.1	X	X	1,297	30	1.5R41	60,462	186	886	64
0.22	260,506	1,637	2,698	307	3.0.1	2,268,058	11,192	24,150	4,225	2.2.0	X	X	1,055	0	1.5R5	63,166	185	1,049	140
0.24	268,848	4,803	6,620	591	3.0.2	2,272,852	11,252	49,758	10,000	2.3.0	X	X	1,005	4	1.6R1	67,421	224	299	99
0.26	316,794	1,841	393	84	3.2	3,271,516	15,153	2,745	550	2.3.1	X	X	316	0	1.6R2	67,623	229	34	3
0.26.2	316,971	1,841	1,198	276	3.2.1	3,284,732	15,176	11,854	4,078	2.3.2	X	X	9,281	41	1.6R3	69,307	230	37	1
–	–	–	–	–	3.2.2	3,286,300	15,184	10,682	2,137	3.0.0	X	X	360	6	1.6R4	69,323	230	185	156
–	–	–	–	–	3.3	3,752,212	17,162	7,386	1,822	3.0.1	X	X	2,787	13	1.6R5	69,758	230	310	115
–	–	–	–	–	3.3.1	3,756,164	17,167	40,314	14,915	3.0.2	X	X	732	2	1.6R6	79,406	270	19	4
–	–	–	–	–	–	–	–	–	–	3.0.3	X	X	2,062	23	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	3.0.4	X	X	1,558	4	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	3.0.5	X	X	232	1	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	3.1.0	X	X	190	0	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–	3.1.1	X	X	1,568	15	–	–	–	–	–

C.2 Detailed Number of Classes Participating to Antipatterns per Releases

Table C.2 – ArgoUML: summary of the number of classes participating to antipatterns in the analysed releases.

Release	AntiSingleton	Blob	ClassDataShouldBePrivate	ComplexClass	LargeClass	LazyClass	LongMethod	LongParameterList	MessageChain	RefusedParentBequest	SpaghettiCode	SpeculativeGenerality	SwissArmyKnife
0.10.1	352	26	136	42	56	16	172	195	79	105	9	0	0
0.12	322	49	140	53	77	19	197	224	92	137	9	0	0
0.14	420	73	158	63	91	47	259	252	228	229	14	0	0
0.16	347	74	157	62	79	44	249	244	260	198	15	0	0
0.18.1	246	71	62	92	112	49	252	219	140	371	7	0	0
0.20	242	84	47	94	144	51	256	222	146	403	13	0	0
0.22	267	80	73	93	129	26	266	233	155	460	20	0	0
0.24	274	84	44	99	135	36	288	248	151	493	19	0	0
0.26	3	116	51	103	166	42	346	299	166	574	22	0	0
0.26.2	3	116	51	103	166	44	348	300	166	574	22	0	0

Table C.3 – Eclipse: summary of the number of classes participating to antipatterns in the analysed releases.

Release	AntiSingleton	Blob	ClassDataShouldBePrivate	ComplexClass	LargeClass	LazyClass	LongMethod	LongParameterList	MessageChain	RefusedParentBequest	SpaghettiCode	SpeculativeGenerality	SwissArmyKnife
1.0	330	600	382	511	1	2403	2372	1087	1043	397	2	54	67
2.0	478	828	412	692	1	2889	3375	2038	1438	692	1	70	43
2.1.1	615	991	448	811	1	3428	4165	2076	1558	695	1	87	38
2.1.2	617	994	450	814	1	3428	4167	2076	1558	694	1	87	38
2.1.3	618	995	454	814	1	3430	4197	2076	1559	694	1	87	38
3.0	975	1417	1453	1137	4	4667	5367	2770	1803	1248	1	146	68
3.0.1	980	1419	1457	1128	4	4673	5370	2775	1816	1248	1	146	68
3.0.2	980	1419	1457	1125	4	4671	5372	2775	1815	1250	1	146	68
3.2	1528	1920	2064	1768	4	7034	6766	2352	2673	2190	0	200	78
3.2.1	1536	1925	2065	1768	4	7044	6779	2357	2693	2191	0	200	79
3.2.2	1536	1931	2066	1767	4	7051	6777	2361	2694	2196	0	200	80
3.3	1783	2194	2281	2123	8	8560	7925	3070	3038	2584	1	228	95
3.3.1	1784	2194	2285	2125	8	8561	7956	3233	3041	2582	1	228	96

Table C.4 – Mylyn: summary of the number of classes participating to antipatterns in the analysed releases.

Release	AntiSingleton	Blob	ClassDataShouldBePrivate	ComplexClass	LargeClass	LazyClass	LongMethod	LongParameterList	MessageChain	RefusedParentBequest	SpaghettiCode	SpeculativeGenerality	SwissArmyKnife
1.0.1	4	40	61	29	43	2	134	43	70	45	12	0	1
2.0M1	5	46	64	33	51	5	167	66	73	54	14	0	0
2.0M2	7	45	66	34	52	4	167	67	81	61	15	0	0
2.0M3	10	56	71	42	61	2	188	74	87	71	16	0	0
2.0.0	9	48	55	34	52	4	153	40	75	70	14	0	0
2.1	9	55	57	40	59	4	169	39	104	65	18	0	0
2.2.0	10	61	66	41	64	4	180	40	112	83	20	0	0
2.3.0	12	68	77	46	67	4	196	43	131	112	24	0	0
2.3.1	12	68	77	46	67	4	196	43	134	112	24	0	0
2.3.2	12	68	77	46	67	4	196	43	134	112	24	0	0
3.0.0	70	71	68	47	67	66	253	90	151	168	36	0	0
3.0.1	70	71	69	47	67	66	258	90	153	167	36	0	0
3.0.2	70	72	68	48	68	71	261	90	157	172	35	0	0
3.0.3	70	72	68	48	67	71	262	91	153	172	35	0	0
3.0.4	70	72	68	48	67	71	262	91	153	172	35	0	0
3.0.5	70	72	68	48	67	71	258	91	153	172	35	0	0
3.1.0	127	93	183	72	99	18	349	95	181	290	39	0	0
3.1.1	127	93	183	72	99	18	349	95	181	290	39	0	0

Table C.5 – Rhino: summary of the number of classes participating to antipatterns in the analysed releases.

Release	AntiSingleton	Blob	ClassDataShouldBePrivate	ComplexClass	LargeClass	LazyClass	LongMethod	LongParameterList	MessageChain	RefusedParentBequest	SpaghettiCode	SpeculativeGenerality	SwissArmyKnife
1.4R3	16	0	4	6	9	4	14	9	20	5	0	0	0
1.5R1	9	0	28	10	12	4	17	15	35	9	0	0	0
1.5R2	12	0	33	13	15	5	23	27	44	14	1	0	0
1.5R3	12	0	33	14	16	5	24	26	44	15	1	0	0
1.5R4	10	0	29	14	15	7	32	28	40	16	4	0	0
1.5R41	10	0	29	14	15	7	32	28	40	16	4	0	0
1.5R5	3	0	23	12	15	11	28	8	42	9	5	0	0
1.6R1	1	0	22	13	16	8	31	8	52	13	6	0	0
1.6R2	1	0	15	13	16	7	31	8	55	14	5	0	0
1.6R3	1	0	15	13	17	7	31	8	52	14	5	0	0
1.6R4	1	0	15	13	17	7	31	8	52	14	5	0	0
1.6R5	1	0	15	13	17	7	31	8	52	14	5	0	0
1.6R6	1	0	17	14	19	9	35	8	66	11	2	0	0

C.3 Detailed Logistic Regression Results

Table C.6 – ArgoUML: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one change.

	Release	DC	DNC	NDC	NDNC	p-value	odds ratio
1	0.10.1	492	93	235	631	0.00	14.17
2	0.12	429	183	227	694	0.00	7.16
3	0.14	494	341	192	825	0.00	6.22
4	0.16	714	84	362	676	0.00	15.84
5	0.18.1	462	309	145	972	0.00	10.00
6	0.20	751	31	515	565	0.00	26.54
7	0.22	711	144	385	689	0.00	8.83
8	0.24	744	159	252	831	0.00	15.40
9	0.26	144	848	51	1195	0.00	3.98
10	0.26.2	282	711	69	1176	0.00	6.75

Table C.7 – ArgoUML: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one bug fixing.

	Release	DB	DNB	NDB	NDNB	p-value	odds ratio
1	0.10.1	37	548	13	853	0.00	4.43
2	0.12	37	575	12	909	0.00	4.87
3	0.14	102	733	8	1009	0.00	17.53
4	0.16	127	671	29	1009	0.00	6.58
5	0.18.1	90	681	27	1090	0.00	5.33
6	0.20	97	685	30	1050	0.00	4.95
7	0.22	43	812	6	1068	0.00	9.42
8	0.24	70	833	39	1044	0.00	2.25
9	0.26	19	973	3	1243	0.00	8.08
10	0.26.2	99	894	14	1231	0.00	9.73

Tables C.6, C.8, C.10, and C.12 report the results of the logistic regression for the correlations between change-proneness and the antipatterns.

Table C.8 – Eclipse: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one change.

	Release	DC	DNC	NDC	NDNC	p-value	odds ratio
1	1.0	1958	1691	501	488	0.10	1.13
2	2.0	3543	1352	897	257	0.00	0.75
3	2.1.1	2177	3735	240	1067	0.00	2.59
4	2.1.2	2346	3568	413	894	0.00	1.42
5	2.1.3	2869	3049	589	718	0.03	1.15
6	3.0	3190	4726	909	1186	0.01	0.88
7	3.0.1	5895	2030	1615	480	0.01	0.86
8	3.0.2	5476	2449	1502	595	0.02	0.89
9	3.2	1682	8905	252	2926	0.00	2.19
10	3.2.1	2609	7988	460	2730	0.00	1.94
11	3.2.2	3045	7558	685	2507	0.00	1.47
12	3.3	1731	10404	192	2804	0.00	2.43
13	3.3.1	4207	7946	812	2174	0.00	1.42

Table C.9 – Eclipse: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one issue fixing.

	Release	DB	DNB	NDB	NDNB	p-value	odds ratio
1	1.0	493	3156	105	884	0.02	1.32
2	2.0	754	4141	120	1034	0.00	1.57
3	2.1.1	425	5487	57	1250	0.00	1.70
4	2.1.2	385	5529	44	1263	0.00	2.00
5	2.1.3	584	5334	67	1240	0.00	2.03
6	3.0	529	7387	58	2037	0.00	2.52
7	3.0.1	944	6981	136	1959	0.00	1.95
8	3.0.2	2391	5534	396	1701	0.00	1.86
9	3.2	778	9809	90	3088	0.00	2.72
10	3.2.1	1234	9363	181	3009	0.00	2.19
11	3.2.2	1025	9578	158	3034	0.00	2.05
12	3.3	1032	11103	85	2911	0.00	3.18
13	3.3.1	1755	10398	361	2625	0.00	1.23

Table C.10 – Mylyn: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one change.

	Release	DC	DNC	NDC	NDNC	p-value	odds ratio
1	1.0.1	187	100	192	1082	0.00	10.51
2	2.0M1	83	270	39	1319	0.00	10.37
3	2.0M2	196	170	187	1199	0.00	7.38
4	2.0M3	400	4	480	995	0.00	206.60
5	2.0.0	218	119	152	1179	0.00	14.17
6	2.1	173	188	107	1269	0.00	10.89
7	2.2.0	188	203	109	1309	0.00	11.10
8	2.3.0	281	169	214	1268	0.00	9.83
9	2.3.1	79	372	40	1445	0.00	7.66
10	2.3.2	396	55	338	1147	0.00	24.38
11	3.0.0	56	579	16	1565	0.00	9.45
12	3.0.1	485	151	388	1192	0.00	9.85
13	3.0.2	129	518	71	1516	0.00	5.31
14	3.0.3	345	303	194	1396	0.00	8.18
15	3.0.4	222	426	193	1397	0.00	3.77
16	3.0.5	58	588	31	1561	0.00	4.96
17	3.1.0	56	827	12	1868	0.00	10.53
18	3.1.1	206	677	97	1784	0.00	5.59

Table C.11 – Mylyn: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one bug fixing.

	Release	DB	DNB	NDB	NDNB	p-value	odds ratio
1	1.0.1	26	261	12	1262	0.00	10.45
2	2.0M1	9	344	2	1356	0.00	17.70
3	2.0M2	5	361	0	1386	0.00	>>300

Table C.12 – Rhino: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one change.

	Release	DC	DNC	NDC	NDNC	p-value	odds ratio
1	1.4R3	36	6	42	74	0.00	10.41
2	1.5R1	56	6	46	90	0.00	17.98
3	1.5R2	37	48	11	251	0.00	17.37
4	1.5R3	65	16	55	215	0.00	15.71
5	1.5R4	38	57	11	270	0.00	16.19
6	1.5R41	67	28	20	261	0.00	30.71
7	1.5R5	61	24	40	247	0.00	15.51
8	1.6R1	71	23	32	260	0.00	24.73
9	1.6R2	20	75	6	288	0.00	12.69
10	1.6R3	24	70	5	294	0.00	19.95
11	1.6R4	84	10	60	239	0.00	33.05
12	1.6R5	59	35	23	276	0.00	19.97
13	1.6R6	27	75	6	346	0.00	20.56

Table C.13 – Rhino: contingency table and Fisher test results for classes that participated in at least one antipattern/underwent at least one bug fixing.

	Release	DB	DNB	NDB	NDNB	p-value	odds ratio
1	1.4R3	31	11	35	81	0.00	6.44
2	1.5R1	20	42	2	134	0.00	31.29
3	1.5R2	7	78	0	262	0.00	Inf
4	1.5R3	58	23	41	229	0.00	13.93
5	1.5R4	3	92	1	280	0.05	9.06
6	1.5R41	29	66	4	277	0.00	30.05
7	1.5R5	27	58	12	275	0.00	10.57
8	1.6R1	32	62	5	287	0.00	29.26
9	1.6R2	0	95	1	293	1.00	0.00
10	1.6R3	1	93	0	299	0.24	Inf
11	1.6R4	80	14	59	240	0.00	23.00
12	1.6R5	39	55	15	284	0.00	13.29
13	1.6R6	3	99	0	352	0.01	Inf

Table C.14 – ArgoUML: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	Proneness to Changes
Antisingleton	8
Blob	2
ClassDataShouldBePrivate	3
ComplexClass	2
LargeClass	2
LazyClass	5
LongMethod	10
LongParameterList	9
MessageChains	10
RefusedParentRequest	9
SpeculativeGenerality	–
SwissArmyKnife	–

Table C.15 – ArgoUML: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	Proneness to Faults
Antisingleton	5
Blob	1
ClassDataShouldBePrivate	2
ComplexClass	–
LargeClass	3
LazyClass	–
LongMethod	1
LongParameterList	5
MessageChains	7
RefusedParentRequest	4
SpeculativeGenerality	–
SwissArmyKnife	–

Table C.16 – Eclipse: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	Proneness to Changes
AntiSingleton	5
Blob	8
ClassDataShouldBePrivate	7
ComplexClass	12
LargeClass	–
LazyClass	12
LongMethod	12
LongParameterList	10
MessageChains	12
RefusedParentBequest	6
SpeculativeGenerality	3
SwissArmyKnife	6

Table C.17 – Eclipse: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between issue-proneness and kinds of antipatterns.

Antipatterns	Proneness to Faults
AntiSingleton	13
Blob	7
ClassDataShouldBePrivate	7
ComplexClass	13
LargeClass	–
LazyClass	12
LongMethod	13
LongParameterList	9
MessageChains	10
RefusedParentBequest	4
SpeculativeGenerality	4
SwissArmyKnife	1

Table C.18 – Mylyn: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	Proneness to Changes
Antisingleton	7
Blob	9
ClassDataShouldBePrivate	9
ComplexClass	2
LargeClass	4
LazyClass	3
LongMethod	17
LongParameterList	7
MessageChains	18
RefusedParentBequest	10
SpeculativeGenerality	6
SwissArmyKnife	–

Table C.19 – Mylyn: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	Proneness to Faults
Antisingleton	–
Blob	–
ClassDataShouldBePrivate	2
ComplexClass	1
LargeClass	–
LazyClass	–
LongMethod	–
LongParameterList	2
MessageChains	1
RefusedParentRequest	1
SpeculativeGenerality	–
SwissArmyKnife	–

Table C.20 – Rhino: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	Proneness to Changes
Antisingleton	–
Blob	1
ClassDataShouldBePrivate	6
ComplexClass	–
LargeClass	4
LazyClass	1
LongMethod	7
LongParameterList	6
MessageChains	13
RefusedParentBequest	6
SpeculativeGenerality	1
SwissArmyKnife	–

Table C.21 – Rhino: number of significant p-values across the analysed releases obtained by logistic regression for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	Proneness to Faults
Antisingleton	–
Blob	–
ClassDataShouldBePrivate	3
ComplexClass	–
LargeClass	3
LazyClass	2
LongMethod	3
LongParameterList	3
MessageChains	7
RefusedParentBequest	–
SpeculativeGenerality	1
SwissArmyKnife	–

Tables C.14, and C.15, and C.16, and C.17, and C.18, and C.19, and C.20, and C.21 report the results of the the number of significant p-values across the analysed releases obtained by logistic regression for the correlations between change-/fault-proneness and kinds of antipatterns.

Table C.22 – ArgoUML: logistic regression results for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	0.10.1	0.12	0.14	0.16	0.18.1	0.20	0.22	0.24	0.26	0.26.2
Antisingleton	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	0.62	0.98
Blob	0.25	0.60	**0.01**	**0.05**	0.61	0.23	0.83	0.30	0.31	0.56
ClassDataShouldBePrivate	**< 0.01**	**< 0.01**	0.06	0.37	**< 0.01**	0.11	0.07	0.63	0.26	0.93
ComplexClass	0.98	0.44	0.75	0.12	0.74	0.97	**0.01**	0.50	**< 0.01**	0.31
LargeClass	0.98	0.21	0.21	0.54	0.06	0.97	**0.01**	0.07	0.62	**0.01**
LazyClass	**< 0.01**	0.11	0.38	**0.01**	**< 0.01**	0.97	**< 0.01**	**0.01**	0.39	0.77
LongMethod	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	**0.02**	**0.01**	0.14	**< 0.01**	**< 0.01**	**< 0.01**
MessageChains	**< 0.01**	**0.04**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
RefusedParentRequest	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	0.64	**< 0.01**
SpeculativeGenerality	0.98	0.36	0.80	0.11	0.11	0.99	0.15	0.14	0.59	0.43
SwissArmyKnife	–	–	–	–	–	–	–	–	–	–

Table C.23 – ArgoUML: logistic regression results for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	0.10.1	0.12	0.14	0.16	0.18.1	0.20	0.22	0.24	0.26	0.26.2
Antisingleton	0.61	0.89	0.08	< **0.01**	< **0.01**	< **0.01**	**0.03**	**0.02**	1.00	0.33
Blob	0.86	0.56	**0.03**	0.87	0.42	0.22	0.32	0.24	0.52	0.13
ClassDataShouldBePrivate	**0.02**	0.10	0.21	0.24	0.63	0.30	0.39	0.32	**0.02**	0.29
ComplexClass	0.32	0.51	0.09	0.83	0.75	0.76	0.63	0.62	0.17	0.90
LargeClass	0.51	< **0.01**	0.36	0.66	0.43	**0.04**	0.09	0.10	0.46	< **0.01**
LazyClass	0.99	0.99	0.98	0.49	0.88	0.84	0.99	0.72	0.99	0.68
LongMethod	0.28	0.79	0.10	0.07	0.29	0.52	0.37	0.99	0.87	< **0.01**
LongParameterList	0.26	0.89	0.27	0.51	0.70	< **0.01**	< **0.01**	**0.01**	**0.01**	< **0.01**
MessageChains	0.64	**0.04**	**0.02**	< **0.01**	< **0.01**	< **0.01**	0.11	**0.05**	0.07	< **0.01**
RefusedParentRequest	< **0.01**	0.81	< **0.01**	< **0.01**	0.58	0.56	0.27	0.12	0.25	< **0.01**
SpeculativeGenerality	0.99	0.99	0.99	0.98	0.98	0.89	0.53	0.98	0.99	0.62
SwissArmyKnife	–	–	–	–	–	–	–	–	–	–

Table C.24 – Eclipse: logistic regression results for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	0.51	0.27	**< 0.01**	0.76	**< 0.01**	0.71	0.17	0.48	**< 0.01**	**< 0.01**	0.11	**< 0.01**	0.12
Blob	**< 0.01**	**0.01**	0.09	**< 0.01**	0.46	**0.02**	**< 0.01**	**< 0.01**	0.58	0.07	0.76	**< 0.01**	**< 0.01**
ClassDataShouldBePrivate	0.08	**< 0.01**	0.44	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	0.23	0.29	**< 0.01**	0.31	0.10	**< 0.01**
ComplexClass	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	0.06	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass	0.95	0.95	0.95	0.93	0.93	0.80	0.99	0.87	0.79	0.42	0.91	0.60	0.43
LazyClass	**0.03**	0.20	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongMethod	**< 0.01**	0.37	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList	**< 0.01**	**< 0.01**	**< 0.01**	0.37	**< 0.01**	**< 0.01**	**< 0.01**	0.08	0.97	**< 0.01**	**0.02**	**< 0.01**	**0.05**
MessageChains	**< 0.01**	0.64	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
RefusedParentBequest	0.10	0.06	1.00	0.73	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**0.01**	0.23	0.62	0.07	**0.01**
SpeculativeGenerality	**0.04**	0.64	0.05	0.09	0.80	0.13	0.15	0.47	0.17	**0.05**	0.18	0.27	**0.04**
SwissArmyKnife	**0.03**	0.88	0.08	**0.03**	0.74	**0.01**	**0.02**	**0.01**	0.05	0.29	0.81	0.76	**< 0.01**

Table C.25 – Eclipse: logistic regression results for the correlations between issue-proneness and kinds of antipatterns.

Antipatterns	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.04**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**
Blob	0.85	**< 0.01**	**0.01**	**0.01**	0.13	0.96	**< 0.01**	**< 0.01**	0.09	0.89	**0.01**	**< 0.01**	0.24
ClassDataShouldBePrivate	**< 0.01**	**< 0.01**	**0.04**	**< 0.01**	0.17	0.98	**< 0.01**	**< 0.01**	0.39	0.81	**0.01**	0.17	0.07
ComplexClass	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass	0.95	0.96	0.96	0.98	0.96	0.97	0.69	0.45	0.78	0.95	0.94	0.61	0.81
LazyClass	0.08	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongMethod	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	0.91	0.18	**< 0.01**	**0.04**	0.67	**< 0.01**	0.05	**0.02**
MessageChains	0.10	**< 0.01**	0.48	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	0.22	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
RefusedParentBequest	**< 0.01**	**0.05**	0.10	**< 0.01**	0.40	**< 0.01**	0.52	0.11	0.76	0.50	0.24	0.39	0.45
SpeculativeGenerality	0.77	0.18	0.15	0.88	0.68	0.09	**0.01**	**0.01**	**< 0.01**	**< 0.01**	0.36	0.15	0.34
SwissArmyKnife	0.44	0.50	0.89	0.57	0.22	0.44	0.73	0.12	0.49	0.10	0.62	0.83	**0.01**

Table C.26 – Mylyn: logistic regression results for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	1.0.1	2.0M1	2.0M2	2.0M3	2.0.0	2.1	2.2.0	2.3.0	2.3.1	2.3.2	3.0.0	3.0.1	3.0.2	3.0.3	3.0.4	3.0.5	3.1.0	3.1.1
Antisingleton	0.99	0.70	0.99	0.52	0.59	0.61	0.12	0.05	0.13	0.86	**0.04**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	**0.04**	0.08	< **0.01**
Blob	0.18	0.29	0.99	0.61	**0.02**	< **0.01**	0.09	**0.01**	0.12	< **0.01**	0.44	< **0.01**	< **0.01**	**0.03**	**0.04**	0.08	0.50	< **0.01**
ClassDataShouldBePrivate	< **0.01**	< **0.01**	0.98	**0.03**	< **0.01**	< **0.01**	0.29	< **0.01**	0.98	< **0.01**	0.14	< **0.01**	0.20	< **0.01**	0.12	0.94	0.97	0.89
ComplexClass	0.77	0.75	1.00	**0.03**	0.40	**0.03**	0.05	0.54	0.70	0.97	0.15	0.10	0.06	0.76	0.54	0.98	0.64	0.17
LargeClass	0.18	< **0.01**	0.99	0.06	0.28	0.54	**0.03**	0.41	**0.02**	0.96	0.89	0.32	< **0.01**	0.35	0.89	0.98	0.60	0.51
LazyClass	0.97	0.07	1.00	0.94	0.11	0.98	0.98	0.98	0.99	0.53	0.77	< **0.01**	0.28	**0.03**	< **0.01**	0.43	0.98	0.97
LongMethod	< **0.01**	< **0.01**	0.98	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
LongParameterList	< **0.01**	0.28	< **0.01**	< **0.01**	0.25	< **0.01**	0.09	**0.02**	< **0.01**	< **0.01**	0.19	0.61	0.22	0.98	0.26	0.86	0.20	0.10
MessageChains	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
RefusedParentBequest	0.61	**0.04**	0.99	< **0.01**	0.31	< **0.01**	0.49	< **0.01**	0.36	< **0.01**	0.61	< **0.01**	0.23	< **0.01**	< **0.01**	0.62	< **0.01**	< **0.01**
SpeculativeGenerality	< **0.01**	0.16	0.99	< **0.01**	0.98	0.25	0.46	0.06	0.97	**0.01**	0.19	< **0.01**	0.12	**0.03**	< **0.01**	0.23	0.67	0.47
SwissArmyKnife	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Table C.27 – Mylyn: logistic regression results for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	1.0.1	2.0M1
Antisingleton	0.99	1.00
Blob	0.15	0.59
ClassDataShouldBePrivate	**0.03**	$<$ **0.01**
ComplexClass	**0.04**	0.68
LargeClass	0.13	0.58
LazyClass	1.00	1.00
LongMethod	0.07	0.58
LongParameterList	0.13	**0.02**
MessageChains	$<$ **0.01**	0.74
RefusedParentRequest	0.64	0.99
SpeculativeGenerality	0.99	1.00
SwissArmyKnife	–	–

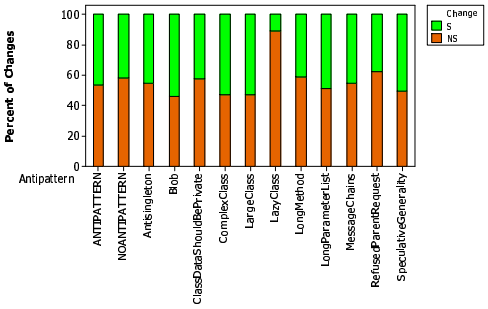
Table C.28 – Rhino: logistic regression results for the correlations between change-proneness and kinds of antipatterns.

Antipatterns	1.4R3	1.5R1	1.5R2	1.5R3	1.5R4	1.5R41	1.5R5	1.6R1	1.6R2	1.6R3	1.6R4	1.6R5	1.6R6
Antisingleton	0.99	0.36	0.06	0.68	0.68	0.53	0.45	1.00	1.00	1.00	1.00	1.00	0.99
Blob	–	–	–	–	–	–	–	–	–	–	–	–	–
ClassDataShouldBePrivate	1.00	0.09	**0.01**	**< 0.01**	0.32	**< 0.01**	**< 0.01**	**< 0.01**	0.13	0.66	**0.04**	0.07	0.18
ComplexClass	1.00	1.00	0.31	0.99	0.99	0.99	0.99	0.87	1.00	1.00	1.00	0.39	0.70
LargeClass	1.00	1.00	**0.05**	0.99	0.99	0.98	0.99	**0.03**	1.00	1.00	0.99	**0.01**	**0.03**
LazyClass	1.00	1.00	0.99	0.09	0.99	0.82	0.40	0.31	0.99	0.99	**0.04**	0.98	0.14
LongMethod	0.69	0.99	0.51	0.44	0.29	0.05	**0.04**	**< 0.01**	0.11	0.17	**< 0.01**	**< 0.01**	**0.02**
LongParameterList	0.99	0.99	**< 0.01**	**< 0.01**	0.30	**< 0.01**	0.98	0.99	0.61	0.33	0.99	0.72	0.52
MessageChains	0.99	**0.01**	**< 0.01**	**0.03**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
RefusedParentBequest	0.99	**0.04**	0.79	0.51	0.62	**< 0.01**	**0.03**	**< 0.01**	0.99	0.99	0.99	**< 0.01**	0.28
SpeculativeGenerality	–	–	1.00	0.99	0.10	0.16	0.59	0.29	1.00	1.00	0.99	**0.01**	0.99
SwissArmyKnife	–	–	–	–	–	–	–	–	–	–	–	–	–

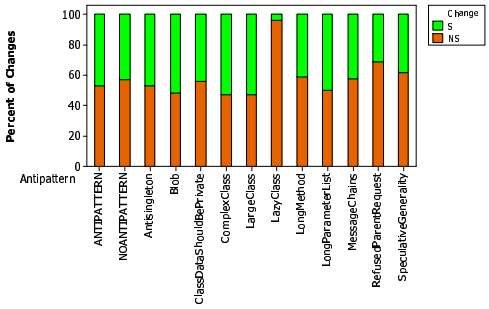
Table C.29 – Rhino: logistic regression results for the correlations between fault-proneness and kinds of antipatterns.

Antipatterns	1.4R3	1.5R1	1.5R2	1.5R3	1.5R4	1.5R41	1.5R5	1.6R1	1.6R2	1.6R3	1.6R4	1.6R5	1.6R6
Antisingleton	0.10	0.21	0.20	0.43	0.40	0.68	0.63	1.00	1.00	1.00	1.00	1.00	1.00
Blob	–	–	–	–	–	–	–	–	–	–	–	–	–
ClassDataShouldBePrivate	0.99	**0.04**	0.20	0.06	0.61	0.57	0.71	**0.02**	1.00	1.00	**0.03**	0.34	1.00
ComplexClass	0.24	0.18	1.00	0.99	1.00	1.00	0.46	1.00	1.00	1.00	1.00	0.42	1.00
LargeClass	0.55	**0.01**	1.00	0.99	1.00	1.00	$<$ **0.01**	1.00	1.00	1.00	0.99	$<$ **0.01**	1.00
LazyClass	0.99	0.99	1.00	0.05	1.00	0.99	0.11	0.99	1.00	1.00	**0.04**	0.99	$<$ **0.01**
LongMethod	0.27	0.11	0.30	0.42	0.75	0.67	0.17	**0.01**	1.00	1.00	**0.01**	$<$ **0.01**	0.15
LongParameterList	0.66	0.06	**0.01**	$<$ **0.01**	0.18	$<$ **0.01**	0.92	0.77	1.00	1.00	0.99	0.09	1.00
MessageChains	**0.05**	0.08	0.16	**0.01**	0.64	**0.02**	$<$ **0.01**	$<$ **0.01**	1.00	1.00	$<$ **0.01**	$<$ **0.01**	0.41
RefusedParentBequest	0.99	0.78	0.84	0.09	1.00	0.36	0.75	0.69	1.00	1.00	0.99	0.45	1.00
SpeculativeGenerality	–	–	1.00	0.99	1.00	0.10	0.38	1.00	1.00	1.00	0.99	**0.01**	1.00
SwissArmyKnife	–	–	–	–	–	–	–	–	–	–	–	–	–

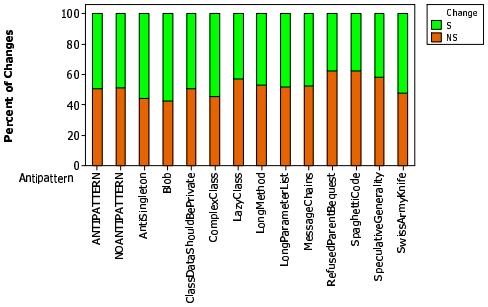
(a) ArgoUML



(b) Mylyn



(c) Eclipse



(d) Rhino

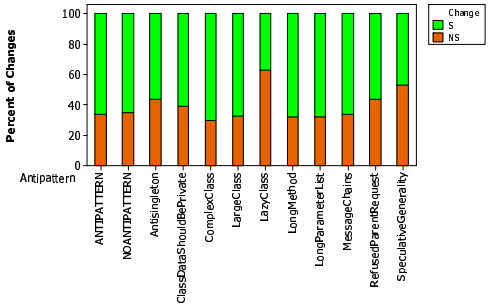


Figure C.1 – Percentages of (S)tructural and (N)on-(S)tructural changes occurring to classes participating (and not) in antipatterns.

Tables C.22, and C.23, and C.24, and C.25, and C.26, and C.27, and C.28, and C.29 provide more details on the results of applying logistic regression for the correlations between changes and faults and kinds of antipatterns.

C.4 Detailed kinds of changes

Figure C.1 shows barcharts of percentages of structural (S) and non-structural (NS) kinds of changes for classes participating or not in different antipatterns, for the whole set of classes participating in any kind of antipattern (ANTIPATTERN) and for the set of classes not participating in any antipattern (NOANTIPATTERN).