Université de Montréal

Evaluating Design Decay during Software Evolution

par
Salima Hassaine

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

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

Août, 2012

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Evaluating Design Decay during Software Evolution

présentée par :

Salima Hassaine

a été évaluée par un jury composé des personnes suivantes :

Président-rapporteur	:	Guy Lapalme
Directeur de recherche	:	Yann-Gaël Guéhéneuc
Codirectrice	:	Sylvie Hamel
Membre du jury	:	Bruno Dufour
Examinateur externe	:	David W. Binkley
Représentant du doyen de la FAS	:	... ...

RÉSUMÉ

Les logiciels sont en constante évolution, nécessitant une maintenance et un développement continus. Ils subissent des changements tout au long de leur vie, que ce soit pendant l'ajout de nouvelles fonctionnalités ou la correction de bogues dans le code. Lorsque ces logiciels évoluent, leurs architectures ont tendance à se dégrader avec le temps et deviennent moins adaptables aux nouvelles spécifications des utilisateurs. Elles deviennent plus complexes et plus difficiles à maintenir. Dans certains cas, les développeurs préfèrent refaire la conception de ces architectures à partir du zéro plutôt que de prolonger la durée de leurs vies, ce qui engendre une augmentation importante des coûts de développement et de maintenance. Par conséquent, les développeurs doivent comprendre les facteurs qui conduisent à la dégradation des architectures, pour prendre des mesures proactives qui facilitent les futurs changements et ralentissent leur dégradation.

La dégradation des architectures se produit lorsque des développeurs qui ne comprennent pas la conception originale du logiciel apportent des changements au logiciel. D'une part, faire des changements sans comprendre leurs impacts peut conduire à l'introduction de bogues et à la retraite prématurée du logiciel. D'autre part, les développeurs qui manquent de connaissances et–ou d'expérience dans la résolution d'un problème de conception peuvent introduire des défauts de conception. Ces défauts ont pour conséquence de rendre les logiciels plus difficiles à maintenir et évoluer. Par conséquent, les développeurs ont besoin de mécanismes pour comprendre l'impact d'un changement sur le reste du logiciel et d'outils pour détecter les défauts de conception afin de les corriger.

Dans le cadre de cette thèse, nous proposons trois principales contributions. La *première contribution* concerne l'évaluation de la dégradation des architectures logicielles. Cette évaluation consiste à utiliser une technique d'appariement de diagrammes, tels que les diagrammes de classes, pour identifier les changements structurels entre plusieurs versions d'une architecture logicielle. Cette étape nécessite l'identification des renommages de classes. Par conséquent, la première étape de notre approche consiste à identifier les renommages de classes durant l'évolution de l'architecture logicielle. Ensuite, la deuxième étape consiste à faire l'appariement de plusieurs versions d'une architecture pour identifier ses parties stables et celles qui sont en dégradation. Nous proposons des algorithmes de bit-vecteur et de clustering pour analyser la correspondance entre plusieurs versions d'une architecture. La troisième étape consiste à mesurer la dégradation de l'architecture durant l'évolution du logiciel. Nous proposons un ensemble de métriques sur les parties stables du logiciel, pour évaluer cette dégradation. La *deuxième contribution* est liée à l'analyse de l'impact des changements dans un logiciel. Dans ce contexte, nous présentons une nouvelle métaphore inspirée de la séismologie pour identifier l'impact des changements. Notre approche considère un changement à une classe comme un tremblement de terre qui se propage dans le logiciel à travers une longue chaîne de classes intermédiaires. Notre approche combine l'analyse de dépendances structurelles des classes et l'analyse de leur historique (les relations de co-changement) afin de mesurer l'ampleur de la propagation du changement dans le logiciel, *i.e.*, comment un changement se propage à partir de la classe modifiée à d'autres classes du logiciel. La *troisième contribution* concerne la détection des défauts de conception. Nous proposons une métaphore inspirée d'un système immunitaire naturel. Comme toute créature vivante, la conception de systèmes est exposée aux maladies, qui sont des défauts de conception. Les approches de détection sont des mécanismes de défense pour les conception des systèmes. Un système immunitaire naturel peut détecter des pathogènes similaires avec une bonne précision. Cette bonne précision a inspiré une famille d'algorithmes de classification, appelés systèmes immunitaires artificiels (AIS), que nous utilisions pour détecter les défauts de conception.

Les différentes contributions ont été évaluées sur des logiciels libres orientés objets et les résultats obtenus nous permettent de formuler les conclusions suivantes:

- Les métriques *Tunnel Triplets Metric* (TTM) et *Common Triplets Metric* (CTM), fournissent aux développeurs de bons indices sur la dégradation de l'architecture. La décroissance de TTM indique que la conception originale de l'architecture s'est dégradée. La stabilité de TTM indique la stabilité de la conception originale, ce qui signifie que le système est adapté aux nouvelles spécifications des utilisateurs.

- La séismologie est une métaphore intéressante pour l'analyse de l'impact des changements. En effet, les changements se propagent dans les systèmes comme les tremblements de terre. L'impact d'un changement est plus important autour de la classe qui change et diminue progressivement avec la distance à cette classe. Notre approche aide les développeurs à identifier l'impact d'un changement.

- Le système immunitaire est une métaphore intéressante pour la détection des défauts de conception. Les résultats des expériences ont montré que la précision et le rappel de notre approche sont comparables ou supérieurs à ceux des approches existantes.

Mots clés: dégradation de l'architecture, impact des changements, défauts de conception.

ABSTRACT

Software systems evolve, requiring continuous maintenance and development. They undergo changes throughout their lifetimes as new features are added and bugs are fixed. As these systems evolved, their designs tend to decay with time and become less adaptable to changing users' requirements. Consequently, software designs become more complex over time and harder to maintain; in some not-so-rare cases, developers prefer redesigning from scratch rather than prolonging the life of existing designs, which causes development and maintenance costs to rise. Therefore, developers must understand the factors that drive the decay of their designs and take proactive steps that facilitate future changes and slow down decay.

Design decay occurs when changes are made on a software system by developers who do not understand its original design. On the one hand, making software changes without understanding their effects may lead to the introduction of bugs and the premature retirement of the system. On the other hand, when developers lack knowledge and–or experience in solving a design problem, they may introduce design defects, which are conjectured to have a negative impact on the evolution of systems, which leads to design decay. Thus, developers need mechanisms to understand how a change to a system will impact the rest of the system and tools to detect design defects.

In this dissertation, we propose three principal contributions. The *first contribution* aims to evaluate design decay. Measuring design decay consists of using a diagram matching technique to identify structural changes among versions of a design, such as a class diagram. Finding structural changes occurring in long-lived, evolving designs requires the identification of class renamings. Thus, the first step of our approach concerns the identification of class renamings in evolving designs. Then, the second step requires to match several versions of an evolving design to identify decaying and stable parts of the design. We propose bit-vector and incremental clustering algorithms to match several versions of an evolving design. The third step consists of measuring design decay. We propose a set of metrics to evaluate this design decay. The *second contribution* is related to change impact analysis. We present a new metaphor inspired from seismology to identify the change impact. In particular, our approach considers changes to a class as an earthquake that propagates through a long chain of intermediary classes. Our approach combines static dependencies between classes and historical co-change relations to measure the scope of change propagation in a system, *i.e.*, how far a change propagation will proceed from a "changed class" to other classes. The *third contribution* concerns design defects detection. We propose a metaphor inspired from a natural immune system. Like any living creature, designs are subject to diseases, which are design defects. Detection approaches are defense mechanisms of designs. A natural immune system can detect similar pathogens with good precision. This good precision has inspired a family of classification algorithms, Artificial Immune Systems (AIS) algorithms, which we use to detect design defects.

The three contributions are evaluated on open-source object-oriented systems and the obtained results enable us to draw the following conclusions:

- Design decay metrics, *Tunnel Triplets Metric* (TTM) and *Common Triplets Metric* (CTM), provide developers useful insights regarding design decay. If TTM decreases, then the original design decays. If TTM is stable, then the original design is stable, which means that the system is more adapted to the new changing requirements.

- Seismology provides an interesting metaphor for change impact analysis. Changes propagate in systems, like earthquakes. The change impact is most severe near the changed class and drops off away from the changed class. Using external information, we show that our approach helps developers to locate easily the change impact.

- Immune system provides an interesting metaphor for detecting design defects. The results of the experiments showed that the precision and recall of our approach are comparable or superior to that of previous approaches.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADvISE	Architectural Decay In Software Evolution
AIS	Artificial Immune Systems
ANOVA	ANalysis Of VAriance
API	Application Programming Interface
AURA	AUtomatic change Rule Assistant
BBN	Bayesian Belief Network
CVS	Concurrent Versions System
DECOR	Defect dEtection for CORrection
FD	Functional Decomposition
IDS	Immune-inspired approach for the Detection of Smells
OO	Object Oriented
PADL	Pattern and Abstract-level Description Language
SC	Spaghetti Code
SVN	apache SubVersioN
UML	Unified Modeling Language

To my parents and grandmother

To my brothers and sister

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. I am deeply grateful for their support. First of all, I would like to thank my supervisor, Yann-Gaël Guéhéneuc, for giving me a chance to select research problems and pursue them. He taught me how to assess the research value of each proposal and how to be selective. He is a patient and thoughtful mentor; he listens to his students, not only what they are saying but also what they are not saying. I am proud to be one of his students. I would also like to thank my co-supervisor, Sylvie Hamel, for her support and advocacy. Her advice as a bioinformatic researcher was very valuable to this dissertation.

My thankful admiration goes to Prof. Giuliano Antoniol, for his advice. He always made time to listen to my research ideas. A great thanks goes also to Prof. Bram Adams for his insightful discussions and his valuable comments on this dissertation.

Special thanks go to Prof. David W. Binkley for enthusiastically accepting to be my external examiner and for reviewing my dissertation. I thank Prof. Bruno Dufour for accepting to be on my committee and for reviewing my dissertation. I thank Prof. Guy Lapalme who have enthusiastically accepted to chair my doctoral committee.

Great thanks to my fellow doctoral students from the Ptidej and Soccer labs for creating an enjoyable working environment. Special thanks go to Fehmi Jaafar, Foutse Khomh, and Segla Kpodjedo. I always enjoy discussing and brainstorming research ideas with you. I would also like to thank all my friends who made the long experience of graduate school more manageable.

Special grateful to my parents Ahmed Hassaine and Soltana Belhandouz for their unconditional love and support, and for believing in me–whenever and wherever. They taught me the importance of passion and persistence in everything I do.

I am also grateful to my grandmother Kheira who always pray for me. Many thanks go to my sister Nacima and my brothers Amine and Aymen for their unlimited love and encouragement.

This research was partially supported by FQRNT, NSERC, and the Research Chairs in Software Patterns and Patterns of Software and in Software Evolution.

— Salima Hassaine

CHAPTER 1

INTRODUCTION

1.1 Research Context

Software systems play a crucial role in modern societies. They are omnipresent from small game applications on smart phones to large embedded systems, such as navigation systems in spacecrafts.

Software systems must evolve to adapt to new requirements to stay useful, else they risk an early death [81, 82]. Therefore, they undergo changes throughout their lifetimes as developers add features, fix defects, or implement changing requirements. As these systems evolve, their designs tend to decay with time and they become less adaptable to new, emerging requirements [28, 122]. Thus, the systems become more complex over time and harder to maintain [12, 55].

Software maintenance is "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [2]. The steps of the maintenance process [92] are: change management, change impact analysis, system release planning, change design, implementation, testing and system release/integration (see Figure 1.1). While the "traditional maintenance" applies only to *corrective maintenance*, which deals with fixing bugs in the code, there are three other types of maintenance that are related to software evolution [117]: *adaptive maintenance* deals with adapting the software to new environments, *perfective maintenance* deals with adding new functionalities to the software according to changes in user requirements, and *preventive maintenance* deals with updating documentation and making the system more maintainable. The long-term effect of corrective, adaptive, and perfective maintenance increases the system's complexity [118]. As systems are continuously changed, their complexity increases and their maintenance cost grow unless preventive maintenance is done to maintain or reduce it.

Over the past two decades, maintenance has been recognised as the most costly and difficult phase in the software life cycle [12, 115]. The maintenance effort has

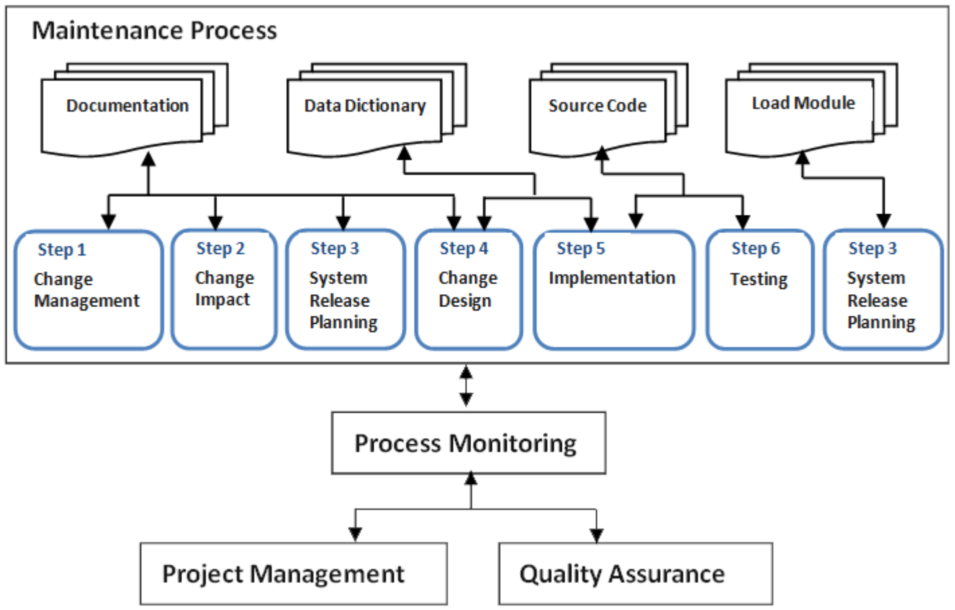


Figure 1.1 – Steps of software maintenance process (From [92]).

been estimated to be frequently more than 70% of the overall software development cost [104]; an increase due in part to *design decay*.

Design decay is a broad term, because a software design can be interpreted either as *architectural design* or *detailed design*. *Architectural design* is "the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relations among them" [1]. It aims at organising the system in components that meet a set of non-functional requirements [11] relying on *architectural styles* [44], *i.e.*, principles of organisation to optimize certain quality requirements, such as the *client-server architecture*. *Detailed design* is concerned by the contents of the identified components [22], *i.e.*, a set of functional requirements relying on *design patterns* [43], which are "good" solutions to recurring design problems. In this dissertation, we use the term *software design* to mean the *detailed design*.

Parnas [99] attributed *design decay* 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 *design decay*. A system that has been repeatedly changed and maintained in this manner is understood by less people over time, *i.e.*, neither the original designers of the system, nor those who made the changes understand the modified system. As a result, the documentation becomes increasingly inaccurate and future changes become more difficult, *i.e.*, they take longer and are more likely to introduce new bugs [99]. Experience [105] shows that 40% of bugs are introduced while correcting previous bugs. Therefore, every new maintenance activity becomes more expensive, because the cost of removing these bugs is high [99].

Van Gurp *et al.* [123] attributed *design decay* to a phenomenon that they called *design erosion*. This phenomenon is due to the cumulative, negative effect of changes on the quality of a system. They suggested that decayed designs make systems more prone to bugs and that, in some not-so-rare cases, a software design and its implementation code must be thrown away because they are too hard to maintain, unless the decay can be stopped before the design is completely unworkable [47, 122, 123]. This phenomenon was observed in many systems [20, 116, 122, 123], such as the Mozilla Web browser: "Netscape was experiencing fierce competition from Microsoft's Internet Explorer. They decided to release their own browser as open source and started working on transforming it into the next generation browser. After half a year of development the developers of the open-source Netscape came to the conclusion that the original Netscape source was eroded beyond repair. They took a major decision and started from scratch" [122]. This example illustrates the difficulty that developers face in maintaining decayed software design and its implementation code. On the one hand, design decay may become an obstacle for further development. On the other hand, fixing design decay may be expensive. In fact, some developers may prefer to redesign from scratch rather than spend effort on trying to fix the existing design. Therefore, developers must understand the factors that drive design decay and take proactive steps that facilitate future changes and ensure that software design does not decay.

Moreton [92] argued that controlling the input of the maintenance process is essential to ensure an effective maintenance. The change management and change impact analysis are the first two steps in the maintenance process (see Figure 1.1). Change management is the identification, estimation, allocation, and monitoring of the resources used to perform a change [1]. Change impact analysis is defined by Pfleeger and Bohner [102] as "the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule", while Arnold and Bohner [17] defined change impact analysis as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change". The software maintenance process can only be optimised if precise and unambiguous information is available about the potential change impact. Experience [99] shows that making changes without understanding their effects can lead to the premature retirement of a system. Therefore, change impact analysis is essential to reduce the amount of corrective maintenance, because fewer bugs will be introduced, and thus *design decay* will be slow down and its effects will be limited.

A major factor affecting the effort required for maintenance is the design quality of systems [134]. Design quality deterioration manifests itself in the form of *design defects*, which are "poor" solutions to recurring software design and implementation problems, such as code smells [40] and design anti-patterns [23]. They occur generally in object-oriented systems when developers lack knowledge and–or experience in solving a design problem or applying some design patterns: "something that looks like a good idea, but which backfires badly when applied" [34]. They are conjectured to have a negative impact on some quality characteristics (*eg.*, change-proneness and fault-proneness [69, 71]) and evolution of systems [23, 40].

Design quality is assessed and improved mainly during formal technical reviews that involve expert inspections. Experience [134] shows that developers spend a lot of time in correcting defects before completing a maintenance task. Thus, the detection of design defects early in the process could substantially reduce the cost of subsequent steps in the development and maintenance phases [104]: designs free of design defects are easier to implement, change, and maintain. However, their manual detection in large designs are highly time- and resource-consuming and error-prone activities [111], because design defects crosscut classes and methods and their descriptions are subject to misinterpretation.

1.2 Problem Statement and Contributions

The above section lead us to formulate our thesis:

Thesis:

Software maintenance is severally impacted by design decay, uncontrolled changes, and design defects. Therefore, to assist developers during software maintenance, we propose to evaluate design decay, to analyse change impact, and to detect design defects.

To verify our thesis, we propose to address three main problems: design decay evaluation, change impact analysis, and design defects detection. To confirm our findings, we propose to use external information, such as bug reports and mailing lists, and to improve the precision and recall of design defects detection.

Problem 1: Design Decay Evaluation

Authors [63, 99, 100, 123, 127] suggested that design decay is the deviation of a software design from the original one. Others [65, 75] suggested stability or resilience as a primary criterion for evaluating a design. However, none of the mentioned studies have been developed to quantify design decay.

To analyse design stability, we must study the traceability of classes across several versions, even if these classes are renamed. Other authors proposed different approaches to analyse the evolution of software designs [73, 75, 130–132]. Most of these previous approaches aim at finding design changes and class renamings occurring in long-lived evolving designs. Identifying class renamings can provide a useful support when a project manager is interested to study the evolution of a set of classes across all the versions of a system. Indeed, if a class disappears in a given release, it could have been renamed or simply deleted. Thus, identifying class renamings is also relevant to study class traceability across versions, which is useful for evaluating design decay. However, existing approaches [75, 132] have somewhat limited performances in time, precision, and recall.

Therefore, our **first contribution** [58, 59] is a novel approach, called $ADvISE$, that exploits a set of metrics to measure design decay during software evolution.

The first step in measuring design decay is to use a diagram matching technique to identify structural changes among versions of a software design. Finding structural changes occurring in long-lived, evolving designs requires the identification of class renamings. Thus, the first step of our approach concerns the identification of class renamings in evolving designs. Then, the second step requires to match several versions of an evolving design to identify decaying and stable parts of a software design. We propose bit-vector and incremental clustering algorithms to match several versions of an evolving design and find stable and decaying micro-designs[1]. The third step consists of measuring the design decay as a whole using stable and decaying micro-designs and we propose a set of metrics that measure design decay. Finally, we performed an empirical study of the impact of bugs and design defects on design decay. We found that decaying designs are more prone to bugs and to design defects than other designs, confirming the importance and usefulness of measures of design decay a posteriori.

Problem 2: Change Impact Analysis

Existing approaches for change impact analysis are based on class dependencies and use static, dynamic, and–or textual analyses [7, 17, 79, 80]. However, in object-oriented systems, the relations between classes make change impact difficult to anticipate because of possible hidden propagation [19]. Historical analysis of data from software repositories [21, 42, 135, 139] provides useful information that complement static and dynamic analyses. Such techniques identify change-impact relations based on the co-changes of software artefacts within a change-set. However, they may fail to capture how changes are spread over "space" (*eg.*, in class diagram) and through a long chain of relations, *i.e.*, they cannot capture how far changes propagate from a given class to the others and whether two co-changing classes are in direct relations or are separated through a long chain of relations. Consequently, they could not help developers prioritise their changes according to the forecast scope of changes.

[1]We use the concept of "micro-design" to mean any subset of a software design, *eg.*, a set of classes and their relations.

Therefore, our **second contribution** [57] is a novel approach to change impact analysis specifically designed to study the scope of change propagation. In Chapter 4, we propose a metaphor between seismology and change impact analysis. In particular, if we analyse the relations chain of a changed class, we may have the intuition that the classes impacted by a change are near the changed class. However, in some cases, the actual classes that must be modified are far away from the changed class. If these classes are not considered before implementing a change, bugs may occur. Our approach considers changes to a class as an earthquake that propagates through a long chain of intermediary classes. Our approach combines static dependencies between classes and historical co-change relations to measure the scope of change propagation in a system, *i.e.*, how far a change propagation will proceed from a "changed class" to other classes.

Problem 3: Design Defects Detection

Several detection approaches [88, 90, 93] detect design defects according to sets of rules and thresholds defined on various metrics. However, threshold values are not easy to define. For example, the Blob and Spaghetti Code design defects (see Appendix B) both describe classes that are too large and too complex. However, a class that is considered large in a given system could be considered average in another. Also, all previous approaches [70, 90, 108] require experts' knowledge and interpretation of the design defects. They focus on detecting one design defect at a time, while some defects share similar characteristics, and exclude classes that are not identical to the defect (given some thresholds). Yet, in the course of our experiments with various detection approaches (based on rules [90], Bayesian Beliefs Networks [70], and B-Splines [108]), we noticed that:

- Several smells have similarities. For example, the Blob and Spaghetti Code antipatterns both describe classes that are too large and too complex; the Blob further describing that these classes should relate to data classes.

- Classes similar but not equal to some design smell are also of interest to developers and quality assurance personnel because they could, in the future, emerge as smells themselves in the "submarine"[2] effect [108].

Moreover, previous approaches have somewhat limited performances in time, precision, and recall.

Therefore, our **third contribution** [56] is a novel approach for design defects detection, called IDS (Immune-based Detection Strategy), based on Artificial Immune Systems, and presented in Chapter 5. A natural immune system protects the body by identifying, learning from, and defending against invading pathogens. Similarly, our contribution is built on the same defense mechanism: a software design is comparable to a body, that we wish to protect from pathogens, such as design defects. Design defect detection approaches are defense mechanisms of the software design. Like pathogens, design defects come in a variety of forms with some defects being only slightly different from others. A natural immune system can handle such similar pathogens with good precision. This good precision is essential for the body and have inspired a family of supervised learning and classification algorithms: Artificial Immune Systems (AIS). We propose to use an AIS algorithm, called Immunos-99 [24], to detect design defects.

1.3 Motivating Scenario

Developers are often concerned with different types of maintenance [117]. In *corrective* maintenance, developers must understand why a previous change introduces a bug and how it can be fixed. In *adaptive* and *perfective* maintenance, developers must understand how to adapt the system to new environments or how to add new functionalities without introducing new bugs. In *preventive* maintenance, developers are often required to detect design defects and correct them to make the system more maintainable and ensure that its design does not decay.

In a typical maintenance scenario, developers are often concerned with implementing new changes in the code. They usually have existing knowledge of the

[2]The term "submarine" was introduced by Oliveto *et al* [108], it is used when several classes may be very close to be identified as antipatterns but remain under the threshold during their evolution.

code or a tip from an expert on the system and know of at least one class that is relevant to the maintenance task. However, developers need help to identify the impact of their changes and locate the rest of the code that is relevant to the maintenance task to make the new changes work properly. When the design is decayed, maintenance costs developers time, effort, and money. Therefore, developers should evaluate the design decay to assess the ratio $cost/benefit$ before implementing new changes.

To illustrate the difficulty developers sometimes face during a maintenance, we outline the main steps involved in a maintenance task:

- **Step 1: Design Decay Evaluation**. First of all, developers should evaluate whether the design of the current system version is decayed or not. For example, the release notes of XercesJ[3], a family of software packages for parsing XML, report that "XercesJ 2.0.0 is a nearly complete rewrite of the XercesJ 1.x code base to make the code cleaner, more modular, and easier to maintain. It includes a completely redesigned and rewritten XML Schema validation engine". This example illustrates the importance of evaluating design decay, because developers can decide to start rewriting another version from scratch instead of working on the current version. Section 3.1 (p. 26) explains how to evaluate design decay.

- **Step 2: Change Impact Analysis**. Once developers decide which classes should be changed they can analyse the impact of their changes, to identify the set of classes that should be modified to accomplish the change. For example, the bug ID200551[4] reports a bug in Rhino[5], an open-source implementation of a JavaScript interpreter, that was introduced by a developer when he implemented a change to class `Kit` and missed a required change to class `DefiningClassLoader`. This example illustrates the importance of identifying the impacted classes before implementing a change. Section 4.2 (p. 68) explains our change impact analysis.

[3]`http://xerces.apache.org/`
[4]`https://bugzilla.mozilla.org/show_bug.cgi?id=200551`
[5]`http://www.mozilla.org/rhino/`

- **Step 3: Design Defects Detection**. Finally, developers should improve the quality of software design by detecting the design defects and correct them. For example, in XercesJ v2.7.0 there are 44 instances of design defects. Section 5.1 (p. 87) explains how to identify these design defects.

In summary, our approaches help developers in evaluating design decay, identifying the change impact, and detecting design defects to ensure an efficient maintenance and to limit the effects of design decay.

1.4 Roadmap

The remainder of this dissertation provides the following content: Chapter 2 (p. 12) reviews related work on design decay, change impact analysis, and design defects detection. Chapter 3 (p. 24) reports our first contribution for evaluating the design decay. Chapter 4 (p. 63) reports our second contribution for studying the scope of change propagation using a seismology metaphor.Chapter 5 (p. 86) reports our third contribution that concerns the design smells detection using an Artificial Immune System. Chapter 6 (p. 98) presents the conclusions of this dissertation and outlines some directions of future research. Appendix A (p. 117) presents the definitions of metrics and quality attributes considered in this dissertation. Appendix B (p. 121) presents the complete list of code smells and antipatterns considered in this dissertation with their definitions.

CHAPTER 2

RELATED WORK

This chapter provides a survey of existing works related to this thesis and identifies the limitations that are addressed by our contributions.

The structure of the chapter is as follows: Section 2.1 provides a description of leading work in design decay evaluation. Section 2.2 discusses the state of the art in change impact analysis. Section 2.3 summarises exiting works in design defects detection.

2.1 Design Decay Evaluation

Design decay is the deviation of actual software design from the original one, *i.e.*, the violation of design choices during evolution [63, 100, 127]. Van Gurp *et al.* [123] suggested that decayed designs make systems more prone to bugs and that, in some not-so-rare cases, a software design and its implementation code must be thrown away because it is too hard to maintain, unless the decay can be stopped before the design is completely unworkable [47]. Macia *et al.* [85] presented a case study on the impact of design antipatterns on design decay.

Perry *et al.* [100] suggested that design decay is due to violation of architecture caused by the process of evolution. Eick *et al.* [37] suggested that a piece of code has decayed if it is more difficult to change than it used to be. Van Gurp *et al.* [123] defined design decay as the cumulative, negative effect of changes on the quality of a software system. Hochstein *et al.* [63] defined design decay as the deviation of actual or concrete design from planned or conceptual design. Williams *et al.* [127] defined design decay as the deviation from the original design. Parnas [99] suggested that the structure of the software system degrades when changes to software are made by people who do not understand the original design concept, because the modifications often invalidate the initial design.

Design decay evaluation relates to two main research directions: design evolution and class renamings detection. Techniques for analysing the design evolution detect structural changes between versions of a software design, typically represented as class diagrams. Identifying class renamings can provide a useful support when a project manager is interested to study the evolution of a set of classes across all the versions of a software system. In fact, even if a class disappears in a given version, it could have been renamed, or simply deleted. Thus, identifying class renamings is also relevant to study class traceability across versions, which is useful for evaluating design decay. The following sections present related work on design evolution and renamings detection techniques.

2.1.1 Software Design Evolution

Antoniol *et al.* [4] proposed an approach that helps its users to deal with inconsistencies by pointing out regions of code that do not match, *i.e.*, added, deleted, and modified classes and methods. They first recovered UML-like class diagrams from the source code of a software system in the Abstract Object Language (AOL). Then, they compared the recovered designs of subsequent software versions using bipartite graph matching. Nodes in the graphs represent the classes of a version and the similarity between two graphs is derived from class and attribute/method names by means of a String Edit Distance. Their approach does not support neither the relations between classes nor the class renamings.

Antoniol *et al.* [5] proposed an automatic approach, based on cosine similarity on class identifiers to automatically identify links between classes obtained from refactoring between two subsequent releases. In particular, the approach aimed at identifying cases of class replacement, split, merge, as well as feature migration from/to other classes. They represented classes of different releases as documents and queries, then applied a vector space model that treats documents and queries as vectors [41], with documents ranked against queries by computing some similarity functions between the corresponding vectors. Their approach does not take into account the relations between classes.

Xing and Stroulia [132] presented the UMLDiff tool to automatically detect structural changes between two versions of a software design. They modeled each design as a directed graph, where nodes are software entities (packages, classes, interfaces, and their fields and methods), and edges represent the relation between them. They used UMLDiff to compare the two directed graphs in terms of additions, removals, moves, renamings, and signature changes of software entities. They also used UMLDiff to study class evolution [131] at the design level to understand phases and styles of evolution [130].

Kpodjedo *et al.* [75] proposed an Error Correcting Graph Matching (ECGM) algorithm to study design diagram evolution. This algorithm is derived from search-based techniques: given two design diagrams D_1 and D_2, the authors aims at finding, among the large set of all possible matchings, a solution that is the best *true match* between classes of D_1 and D_2. They identified evolving classes that maintain a stable structure of relations (association, aggregation, and inheritance) with other classes and that constitute the stable backbone of a software design.

Kimelman *et al.* [73] designed a Bayesian framework to perform diagram matching. They represented diagrams as graphs whose nodes have attributes, such as name, type, connections to other nodes, and containment relations. Probabilistic models are used for rating the quality of candidate correspondences based on various features of the nodes in the graphs. Given the probabilistic models, they can find high-quality correspondences between two diagrams using search algorithms.

2.1.2 Discussion

We share with all the above authors the idea that design decay is the deviation of a software design from the original one. Existing approaches for design evolution compare two versions of a software design to study its evolution. However, to the best of our knowledge, none of the mentioned studies have been developed to automatically quantify design decay. In Chapter 3, we propose an approach to evaluate design decay. The first step in measuring design decay is to use a diagram matching technique to identify structural changes among versions of a software design. Finding structural changes occurring in long-lived, evolving designs requires the identification of class renamings. Then, we propose a set of structural and textual similarity measures to identify class renamings in evolving designs. The second step requires to match several versions of an evolving design. Then, we propose a bit-vector algorithm to match several design versions. The final step is to measure design decay. We propose a set of software metrics that measure design decay: the *Tunnel Triplets Metric* and *Common Triplets Metric*. We use these metrics to study the impact of design defects on design decay.

2.1.3 Renamings Detection

Eshkevari *et al.* [39] presented a study of identifier renamings in software systems, studying how terms (identifier atomic components) change in source code identifiers. They proposed an approach based on the normalized edit distance to detect identifier renamings. Wei *et al.* developed AURA [129], a novel hybrid approach that combines call dependency and text similarity analyses to provide developers with change rules when adapting their systems from one version of a framework to the next. Dagenais *et al.* developed SemDiff [35], a tool that recommends replacements for framework methods that were accessed by a client system and deleted during the evolution of the framework. Schäfer *et al.* [112] mined framework-usage change rules from already-ported instantiations. The three previous approaches compute support and confidence value on call dependency analysis. Godfrey *et al.* [48] presented a semi-automatic hybrid approach to perform origin analysis using text similarity, metrics, and call dependency analyses. Xing and Stroulia [133] developed Diff-CatchUp to analyse textual and structural similarities of UML class diagram to recognise API changes. Kim *et al.* [72] presented an automated approach to infer high-level renaming patterns.

2.1.4 Discussion

The above renamings detection techniques detect renamings at method level and use text-based similarities. Thus, they cannot detect renamed methods that do not have similar names with their target methods. Call dependency-based approaches provide useful information to identify renamed methods that may not be detected by text-based approaches. However, they cannot detect renamed methods for target methods that are not used in frameworks. In Chapter 3, we propose similarity measures to detect renamings at class level. Our approach could also be adapted to detect renamings at method level.

2.2 Change Impact Analysis

Change impact analysis aims at identifying software artefacts being affected by a change; it provides the potential consequences of a change and estimates the set of artefacts that must be modified to accomplish a change [17].

2.2.1 Structure-based Change Impact Analyses

Arnold and Bohner proposed several models of change propagation [7, 17]. These models are based on code dependencies and algorithms, including slicing and transitive closure, to assist in assessing the impact of changes. Dependency analysis of source code is performed using static or dynamic analyses. When performing change impact analysis with call graphs, the impact of a change in a method is the transitive closure of all callers and callees. Therefore, it can be inaccurate, by reporting false candidates that do not change (low precision) and failing to estimate some classes that actually do change because the analysis is restricted to method calls (low recall).

Weiser *et al.* [126] proposed also slicing techniques to determine all the code locations that may affect a reported location of a failure. Static slicing is typically based on data- and control-flow graphs that are computationally expensive to process and analyse and can report large slices [16]. Thus, dynamic slicing [3, 137] and probabilistic slicing [110], have been proposed in the literature to reduce the size of slices. However, their analysis is expensive.

Law *et al.* [80] argued that static slicing is much more precise than transitive closure on call graphs but it may return large sets of classes that are supposed to be impacted by a change. Dynamic slicing can improve the conservative behavior of static slicing. However, it is subject to the risk of lower precision and recall as it depends on the chosen scenarios and–or executed test cases. Consequently, Law *et al.* [80] introduced a new approach to method-level change impact analysis. They used path profiling technique [79] to compress dynamic traces, then they applied PathImpact algorithm to predict dynamic change impact. Their approach can provide potentially more useful predictions of change impact than method-level static slicing in situations where specific system behaviors are the focus.

Rajlich *et al.* [106] described some incremental change activities, such as impact analysis and change propagation, in which programming concepts and program dependencies play a key role. They argued that different kinds of class interactions have different likelihoods of change propagation. **(TODO:Contrary to their approach, we propose to use class interactions to measure change impact...)**

Zaidman *et al.* [136] proposed a technique for uncovering important classes in a system architecture. They used a technique that was originally developed to identify important hubs on the Internet, *i.e.*, pages with many links to "authorative" pages [74]. They verified that important classes in the system correspond to the hubs in the dynamic call-graph of a system.

2.2.2 History-based Change Impact Analyses

Ying *et al.* [135] and Zimmermann *et al.* [139] proposed to mine version-control systems, using association rules, to identify logical couplings [42] between classes. A change occurring in class A may have an impact on another class B if in the past they changed together. Such historical analysis can capture change couplings that cannot be captured by static and dynamic analyses.

Bouktif *et al.* [21] used a technique from speech recognition to infer cause–effect relations from the revision histories. Their approach relies on the technique of dynamic time warping to group files with histories of changes of different lengths. The values of their approach precision and recall are higher than previous approaches [135, 139].

Canfora *et al.* [27] proposed an approach based on information-retrieval techniques to derive the set of classes impacted by a proposed change request. They argued that the histories of change requests is a useful descriptor of classes when it is used for change impact analysis.

German *et al.* [45] proposed a method which determines the impact of previous code changes on a particular code segment based on a change impact graph. Given a location of failure, their method annotates the neighbours of this failure in the graph by marking the recent changes. Thus, it determines all the changed areas of the software system that affect the reported location of a failure.

2.2.3 Probabilistic Approaches

Zhou *et al.* [138] and Mirarab *et al.* [89] presented a change propagation analysis based on Bayesian networks that incorporates static source code dependencies as well as different features extracted from the history of systems and uses a sliding window algorithm to group them. Their change propagation model can predict future change couplings. Mirarab *et al.* [89] used Bayesian belief networks as a probabilistic tool to make such predictions systematically. Their approach mainly relies on dependency metrics calculated using static analysis and change history extracted from a version-control system.

Antoniol *et al.* [6] incorporated static source code dependencies and other features extracted from the release history of a system, such as author information. Then, they applied the LPC/Cepstrum technique to mine a version-control system for classes having evolved in the same or very similar ways. Their approach can find classes having very similar maintenance evolution histories.

Ceccarelli *et al.* [31] proposed the use of a generalisation of univariate autoregression model to capture the evolution and inter-dependencies between multiple time series. They applied the bivariate Granger causality test [49] to infer the mutual dependencies between classes, analysing the time series representing the change histories of a class A to predict the changes of another class B. Their preliminary results showed that change impact relations inferred with the Granger causality test are complementary to those inferred with association rules.

2.2.4 Hybrid Approaches

Girba *et al.* [46] proposed an approach, named Yesterday's Weather, to identify classes that are likely to change in the next version. This approach is based on the retrospective empirical observation that classes that changed the most in the recent history will also undergo important changes in the near future.

Malik and Hassan [86] proposed the use of adaptive change propagation heuristics. These heuristics combine the use of history heuristic, containment heuristic, call use depends heuristic, and code ownership heuristic. The proposed adaptive heuristic uses a best heuristic table to track for each change entity the best heuristic, and uses the development replay framework [61] to measure the performance of that heuristic.

Hassan *et al.* [60] proposed a model of change propagation, based on several heuristics for predicting the set of classes that should change after a particular class has been changed. In their approach, they combined various sources of data for change impact analysis, such as static dependencies between classes and historical co-change relations.

2.2.5 Discussion

Some existing approaches for change impact analysis are based on class dependencies and use static, dynamic, and–or textual analysis [7, 17, 79, 80]. However, in object-oriented systems, the relations between classes make change impact difficult to anticipate because of possible hidden propagation [19]. Historical analysis [21, 42, 135, 139] of data from software repositories provides useful information that complement static and dynamic analyses. Such techniques learn change impact relations based on the co-changes of software artefacts within a change-set. However, they may fail to capture how changes are spread over "space" (*eg.*, among classes in class diagram) and through class levels (*i.e.*, the distance between co-changed classes). Thus, they could not help developers prioritise their changes according to the forecast scope of changes, which can lead to poor effort and cost estimations.

Consequently, none of these approaches have been used to study how far changes propagate from a given class to the others, *i.e.*, whether two co-changing classes are in direct relation or are separated through a long chain of relations. In particular, if we analyse the relation chain of a changed class, we may have the intuition that the classes impacted by a change are near the changed class. However, in some cases, the actual classes that must be modified are far away from the changed class. If these classes are not considered before implementing a change, bugs will occur. In Chapter 4, we propose an approach to change propagation analysis specifically designed to study the scope of change propagation, based on a metaphor between seismology and change impact analysis. We use this approach to study the impact of design defects on change propagation.

2.3 Design Defects

Code smells [40] and antipatterns [23], collectively called in the following design defects, are poor solutions to recurring software design and implementation problems. They occur generally in object-oriented systems when developers lack knowledge and–or experience in solving a design problem or applying some design patterns [43].

2.3.1 Specification of Design Defects

Several books relate to design defects. Webster [125] wrote the first book on anti-patterns in object-oriented development; his contribution covers conceptual, political, coding, and quality-assurance problems. Riel [107] defined 61 heuristics characterising "good" object-oriented programming. These heuristics deal with classes, objects, and relations. They allow developers to assess the quality of their systems manually and provide a basis to improve designs and implementations. Fowler [40] defined 22 code smells that are low-level design defects in the source code of systems, suggesting that developers should apply refactorings. Code smells are described in an informal style and associated with methods to locate them smells through manual inspections of the source code. Mäntylä [87] and Wake [124] proposed classifications of code smells. Brown's book [23] is more focused on the design and implementation of object-oriented systems than Webster's. Brown *et al.* described about 40 anti-patterns textually, which are general object-oriented design defects and include well-known antipatterns, *eg.*, Blob.

These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic and industrial audience. However, it is difficult to build detection and correction algorithms from their textual descriptions, because they lack precision and are prone to misinterpretations. We build upon this work to propose an approach to characterise design defects and identify classes with similar characteristics. We use the term defect to acknowledge that, in certain contexts, a code smell or an antipattern may be unavoidable and the best way to design and–or implement (part of) a system, *eg.*, parsers are often Spaghetti Code.

2.3.2 Detection of Design Defects

Several approaches to specify and detect design defects have been proposed in the literature. They range from manual approaches, based on inspection techniques [120], to metric-based heuristics [88, 90, 93], where smells are detected according to sets of rules and thresholds defined on various metrics. Manual approaches were defined, for example, by Travassos *et al.* [120], who introduced manual inspections and reading techniques to detect code smells.

Marinescu [88] presented a metric-based approach to detect smells with detection strategies, which capture deviations from good design principles and consist of combining metrics with set operators and comparing their values against absolute and relative thresholds. Similarly to Marinescu, Munro [93] proposed metric-based heuristics to detect code smells; the heuristics are derived from template similar to the one used for design patterns [43]. He also performed an empirical study to justify the choice of metrics and thresholds.

Moha *et al.* [90] proposed the DECOR method to specify and automatically generate detection algorithms. DECOR includes a domain-specific language based on a literature review of existing work. It also includes algorithms and a platform to automatically convert specifications into detection algorithms and apply these algorithms on any system. DECOR produces detection algorithm with good precision and perfect recall while allowing quality assurance personnel to adapt the specifications to their context.

Khomh *et al.* [70] argued that threshold-based approaches do not handle the uncertainty of the detection results and, therefore, miss borderline classes, *i.e.*, classes with characteristics of design smells "surfacing" slightly above or "sinking" slightly below the thresholds because of minor variations in their characteristics. Consequently, they proposed a Bayesian Belief Network (BBN) for the detection of design smells in systems, which output is the probability that a class exhibiting the characteristics of a smell be truly a smell. Thus, their approach handles the degree of uncertainty for a class to be a smell. They also showed that BBNs can be calibrated using historical data both from a similar and from a different context.

Oliveto *et al.* [108] proposed ABS, an approach to detect design smells in systems using signatures of the classes and of the smells. The signature of a smell is computed as the average of the signatures of a set of known classes participating to that smell. For each class in a system, using B-splines, they compared the signature of the class with that of a smell and computed their similarity to detect occurrences of the smell. They reported a case study and claimed that ABS outperforms previous approaches in precision and recall while being simpler in practice.

Some visualisation techniques, for example [114], were used to find a compromise between fully-automatic detection techniques, which are efficient but lose track of the context, and manual inspections, which are slow and subjective. Other approaches perform fully-automatic detection and use visualisation to present the detection results [78, 121].

Catal *et al.* [30] used several machine learning algorithms to predict the defective modules. They investigated the effects of dataset size, metrics set, and feature selection techniques for software fault prediction problem. They employed several algorithms based Artificial Immune Systems.

Kessentini *et al.* [68] independently used an Artificial Immune System [24] to estimate the risks of classes to deviate from “normality”, *i.e.*, a set of classes representing a “good” design. They used structural data to describe a design, *i.e.*, classes, fields, methods... They showed that 90% of the more riskiest classes in GanttProject and Xerces are defects.

2.3.3 Impact of Design Defects

Despite the above studies on design defects, only a few studies empirically analysed the impact of design defects on source code-related phenomena, in particular class change- and fault-proneness.

Bois *et al.* [18] showed that the decomposition of God Classes into a number of collaborating classes using well-known refactorings can improve comprehension. They did not consider source code evolution phenomena.

Wei and Shatnawi [84] investigated the relation 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.

Olbrich *et al.* [97] 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.

Khomh *et al.* [69, 71] studied the impact of classes with design defects (code smells and antipatterns) on change-proneness and fault-proneness. They showed that classes participating in design defects are more change- and fault/issue-prone than classes not participating in design defects.

2.3.4 Discussion

Previous approaches advanced the state-of-the-art in the specification and detection of design defects but all require experts' knowledge and interpretation. Moreover, they focus on detecting one kind of design defects at a time, while some design defects are similar and classes with characteristics similar but no identical to some design defects are also of interest to developers and quality assurance personnel. In Chapter 5, we propose an immune-inspired approach for the detection of design defects. We use object-oriented metrics [32] computed on instances of smells as input to our algorithm following our parallel between object-oriented software systems and living bodies. We analyses our approach in two distinct, industrial-like scenarios. We also discuss all the advantages of our approach over previous approaches, including precision and recall.

This previous work raised the awareness of the community towards the impact of code smells and antipatterns on software development. We build on this previous work and propose a more detailed and extensive empirical study of the impact of design defects on design decay phenomena (see Chapter 3).

CHAPTER 3

DESIGN DECAY EVALUATION

Software design plays an important role in software development, because it contains information that eases the communication among stakeholders, such as developers and project managers. Each stakeholder is concerned with different software characteristics[1] that are affected by the design. For example, developers use designs to verify whether their implementation is conform to earlier design decisions. Project managers use designs to create teams and allocate resources among them. Architects are concerned with ensuring that the design meets their design goals [11]. The above examples illustrate the importance of software design during development. Thus, each stakeholder must keep track of the design stability. On top of that, there is often a gap between the design and its implementation [94] from the start, because it is not easy to design software reflecting the intention of developers [64].

Software systems evolve continuously, requiring continuous maintenance and development. They undergo changes throughout their lifetimes as developers add features, fix bugs, or implement changing requirements. As these systems evolve, their designs tend to decay with time and they become less adaptable to new, emerging requirements [28, 122]. Design decay is the deviation of actual software design from the original one, *i.e.*, the violation of design choices during evolution [63, 100, 127]. Van Gurp *et al.* [123] suggested that decayed designs make systems more prone to bugs and that, in some not-so-rare cases, a software design and its implementation code must be thrown away because it is too hard to maintain, unless the decay can be stopped before the design is completely unworkable [47]. Macia *et al.* [85] presented a case study on the impact of design defects on design decay. Their study revealed that 78% of design decay in the software systems were related to design defects [23].

In this chapter, we propose a novel approach, $ADvISE$, that proposes design decay indicators, which could serve as symptoms of decay, in the context of an

[1]We use the term "software characteristics" to mean quality attributes (see Appendix A.2).

evolving design. In our approach, we use the term "design" to mean any structural model of a system, *eg.*, a UML class diagram. $ADvISE$ aims at analysing the evolution of a software design at various abstraction levels to calculate measures of design decay. Our approach uses so-called "triplets", which we define as $T = (C_{Source}, R, C_{Target})$, where C_{Source} and C_{Target} represent two classes and R is a relation between them. We use the concept of "micro-design" to mean any subset of a software design.

The first step in measuring design decay is to use a diagram matching technique to identify structural changes among versions of a software design. Finding structural changes occurring in long-lived, evolving designs requires the identification of class renamings. Thus, a first contribution of this chapter is a set of structural and textual similarity measures to identify class renamings in evolving designs.

The second step requires to match several versions of an evolving design to identify decaying and stable micro-designs. Thus, the second contribution of this chapter is an incremental clustering algorithm to match versions of a design in order to find stable micro-designs ($S\mu_D$) that exist in all versions, and decaying micro-designs ($D\mu_D$), which are represented by the set of "triplets" that are deleted in a given version.

The third step consists of using the previously-identified, stable micro-designs to propose metrics that measure design decay. Thus, the third contribution is a set of software metrics that measure design decay: the *Tunnel Triplets Metric* (TTM) and *Common Triplets Metric* (CTM). These metrics could be used as predictors of bug proneness [123] and design defect proneness [85].

We validate $ADvISE$ by studying the design history of five systems, ArgoUML, DNSjava, JFreeChart, Rhino and XercesJ, and observing when and how their designs decayed. We also use this validation to show that decaying designs are more prone to design defects and to bugs than stable designs, basically confirming the importance and usefulness of measures of design decay a posteriori. Thus, we first answer the following preliminary questions:

- **RQ1: What are the thresholds for class renaming detection?** We show that we can systematically choose adequate thresholds that provide an optimal F-measure (precision and recall) for class renaming detection.

- **RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?** We show that our approach has good precision and recall for class renamings detection.

Then, we answer the following research questions:

- **RQ3: What are signs of design decay and how can they be tracked down?** We show that our design decay metrics (TTM and CTM) provide us useful insights regarding the signs of software aging.
- **RQ4: Do stable and decaying micro-designs have the same risk to be bug-prone?** We show that stable micro-designs, belonging to the original design, are significantly less bug-prone than decaying micro-designs.
- **RQ5: Are decaying micro-designs more prone to design defects than stable micro-designs?** We show that stable micro-designs, belonging to the original design, are significantly less prone to design defects than decaying micro-designs.
- **RQ6: How does ADvISE perform?** We show that the time performance of our approach is good, outlining the execution time of each step of $ADvISE$.

The chapter is organised as follows. Section 3.1 describes our approach. Section 3.2 and 3.3 present five case studies and discuss our approach. Finally, Section 3.4 concludes and outlines future work.

3.1 Approach

This section presents $ADvISE$, our approach to compute metrics of design decay. Figure 3.1 shows an overview of the steps of our approach, which we describe in detail below. Our approach consists of five steps. Given two versions of an object-oriented software system, $ADvISE$ extracts their designs as UML-like class diagrams [50] using an existing tool, PADL[52]. Second, it identifies class renamings using a novel combination of structural and textual similarities. Third, it matches each pair of two subsequent versions of software designs, using a bit-vector algorithm, to identify their stable triplets $T = (C_{Source}, R, C_{Target})$, where C_{Source}

and C_{Target} represent two classes and R is a relation linking them. Fourth, it applies an incremental clustering algorithm to group connected triplets into clusters to find stable ($S\mu_D$) and decaying ($D\mu_D$) micro-designs. Finally, our approach uses the sets of stable triplets between two design versions to compute the TTM and CTM metrics that measure the design decay.

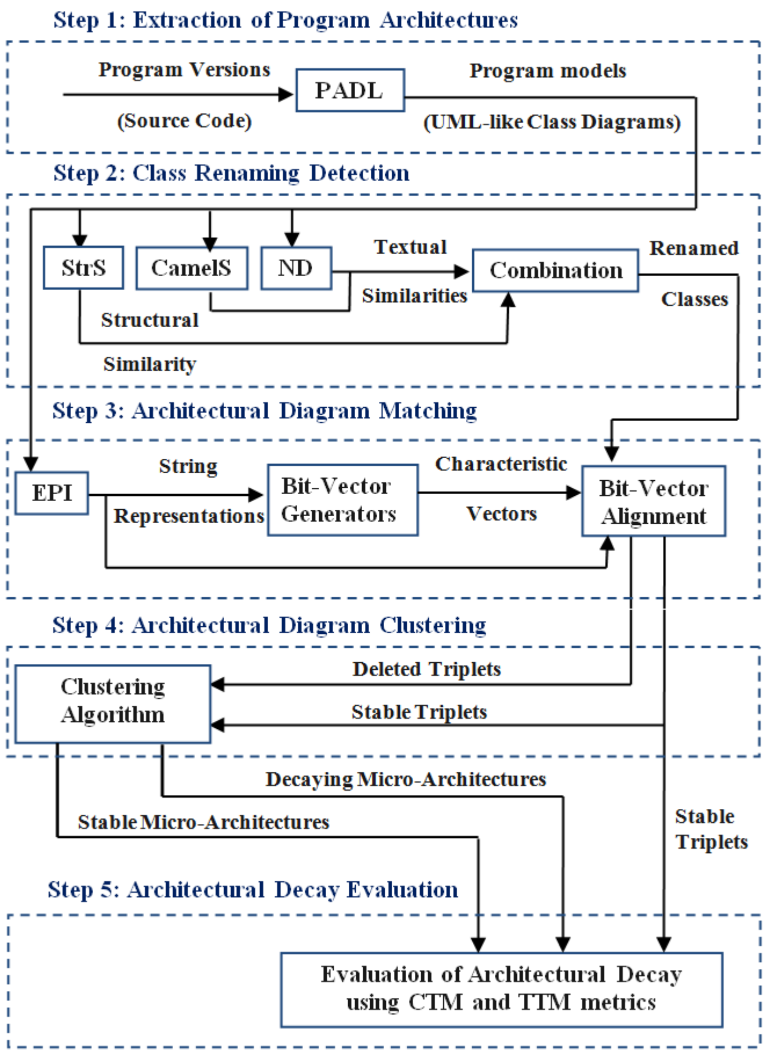

Step 1: Extraction of Program Architectures
Program Versions
(Source Code)
PADL
Program models
(UML-like Class Diagrams)
Step 2: Class Renaming Detection
StrS
CamelS
ND
Textual
Similarities
Combination
Renamed
Classes
Structural
Similarity
Step 3: Architectural Diagram Matching
EPI
String
Representations
Bit-Vector Generators
Characteristic
Vectors
Bit-Vector Alignment
Step 4: Architectural Diagram Clustering
Clustering Algorithm
Deleted Triplets
Stable Triplets
Decaying Micro-Architectures
Stable Micro-Architectures
Stable
Triplets
Step 5: Architectural Decay Evaluation
Evaluation of Architectural Decay using CTM and TTM metrics

Figure 3.1 – Approach Overview.

3.1.1 Step 1: Extraction of Design Diagrams

In our approach, we represent a software design by a reverse-engineered UML-like class diagram [50]. We use an existing tool, PADL [52], to automatically reverse-engineer class diagrams from the source code of object-oriented software systems. The PADL meta-model defines all the constituents required to describe the static structure of software and part of their behaviour, including message sends and binary class relations, such as: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [51]). The PADL tool is associated with several parsers to build models of software from AOL, C++, C#, and Java. A model of a software is represented as a digraph (directed graph) with vertices being the classes and edges representing the relations among classes, as illustrated in Figure 3.2(a).

3.1.2 Step 2: Detection of Class Renaming

The second step in measuring design decay consists of analysing the changes between two subsequent versions V_i and V_{i+1} of a software design, such as: additions, deletions, and renamings of classes. Thus, based on two subsequent versions of class diagrams (obtained in the previous step), we first extract the set of classes that exist in version V_i and disappear in version V_{i+1}. Then, we apply our class renaming detection to assess whether these classes were renamed or deleted. Our approach has the potential to discover two cases of renamings in a fully-qualified class name: (1) Class renaming with or without changing the package name; (2) Package renaming without changing the class name. We use structural and textual metrics to assess the similarities between some original and candidate renamed classes, which we describe in detail in the following.

3.1.2.1 Structural Similarity

We define a structure-based similarity, $StrS$, between a candidate original class C_A and a a candidate renamed class C_B, as the percentage of their common methods, attribute types, and relations (*i.e.*, those having the same target[2]). We assume

[2]We compare six types of logical connections: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [51]).

that two methods M_1 and M_2 are common in C_A and C_B if they have the same signatures (return types, names, modifiers, and parameter list).

Let $S(C_A)$ and $S(C_B)$ be the set of methods, attribute types, and relations of C_A (respectively, C_B). We are inspired by the Jaccard coefficient to quantify similarity between the $S(C_B)$ and $S(C_B)$. The Jaccard coefficient is a measure used for comparing the similarity of sample sets. It provides a percentage of similarity defined as the size of the intersection divided by the size of the union of the sample sets. In our formulas, instead of dividing the size of the intersection by the size of the union, we divide it by the sum of sizes of the two sets and multiply it by 2. Thus, if the two sets are equal their similarity will be equal to 1. The structural similarity of C_A and C_B is computed by comparing $S(C_A)$ to $S(C_B)$ as:

$$StrS(C_A, C_B) = \frac{2 \times |S(C_A) \cap S(C_B)|}{|S(C_A)| + |S(C_B)|} \in [0, 1]$$

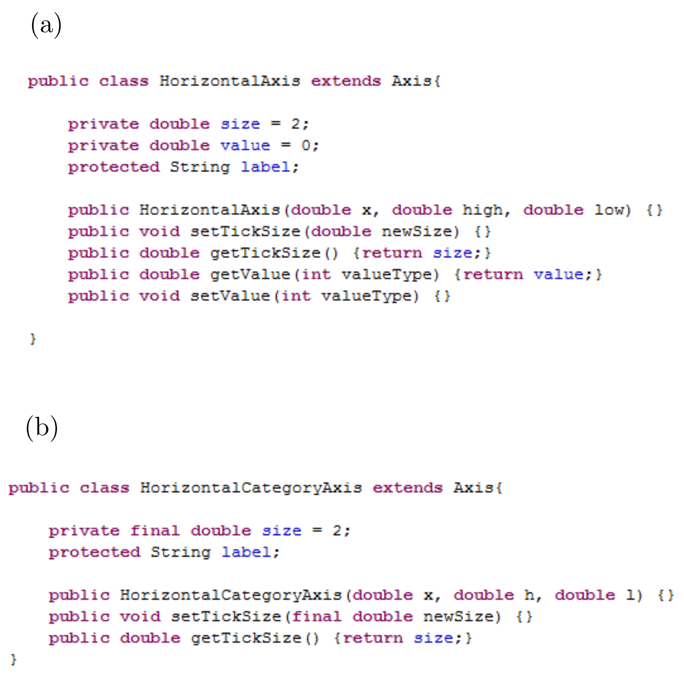
If $StrS(C_A, C_B) = 0$, then C_A and C_B do not have any common methods, attribute types, or relations. If $StrS(C_A, C_B) = 1$, then classes C_A and C_B have the same sets of methods, attribute types, and relations. Given C_A, our algorithm reports C_B with the highest $StrS$ similarity as the best candidate renamed from C_A.

Example 1: Figure 3.2 illustrates the structure of two classes C_A=`HorizontalAxis` and C_B=`HorizontalCategoryAxis`. Let $S(C_A)$ and $S(C_B)$ be the set of methods, attribute types, and relations of C_A (respectively, C_B). $S(C_A) \cap S(C_B) = \{$2 attribute types (`long` and `double`), 1 constructor, 2 methods (`void setTickSize(double)` and `void getTickSize()`), 1 inheritance $\}$. $|S(C_A) \cap S(C_B)| = 6$, $|S(C_A)| = 9$, $|S(C_B)| = 6$. The $StrS$ of C_A and C_B is:

$$StrS(C_A, C_B) = \frac{2 \times 6}{9 + 6} = 0.80$$

3.1.2.2 Textual Similarity

Given an original class C_A, our previous algorithm reports a set of best candidate renamed classes $\{C_{B_1}, ..., C_{B_n}\}$ that have the highest $StrS$ similarity values. We want to select one best candidate renamed class, *i.e.*, the one whose name is the most similar to C_A in addition to having the greater number of common attribute



(a)

```
public class HorizontalAxis extends Axis{

    private double size = 2;
    private double value = 0;
    protected String label;

    public HorizontalAxis(double x, double high, double low) {}
    public void setTickSize(double newSize) {}
    public double getTickSize() {return size;}
    public double getValue(int valueType) {return value;}
    public void setValue(int valueType) {}

}
```

(b)

```
public class HorizontalCategoryAxis extends Axis{

    private final double size = 2;
    protected String label;

    public HorizontalCategoryAxis(double x, double h, double l) {}
    public void setTickSize(final double newSize) {}
    public double getTickSize() {return size;}
}
```

Figure 3.2 – An example of class renaming.

types, methods and relations. Consequently, we compute the textual similarity between the original class C_A and each of the candidate renamed classes C_{B_i} $i \in [1, n]$, using a Camel-Case-based Similarity ($CamelS$) and the Normalised Edit Distance (ND).

We first tokenise the names of C_A and C_B using a Camel Case Splitter, which is the fastest and most widely used identifier splitting algorithm [15], it operates as follows. First, special symbols (such as underscore, pointer access, etc.) are replaced with the space character. Second, identifiers are split where terms are separated using the Camel Case convention. For example, "familyName" is split into "family" and "Name".

Then, we compute $CamelS$ similarity between C_A and C_B as the percentage of common tokens between the names of C_A and C_B. Let $T(C_A)$ (respectively, $T(C_B)$) be the set of tokens in the name of C_A (respectively, name of C_B). We compute the $CamelS$ similarity between C_A and C_B by comparing $T(C_A)$ to $T(C_B)$ as:

$$CamelS(C_A, C_B) = \frac{2 \times |T(C_A) \cap T(C_B)|}{|T(C_A)| + |T(C_B)|} \in [0, 1]$$

If $CamelS(C_A, C_B) = 0$, then the names of C_A and C_B do not have common tokens. If $CamelS(C_A, C_B) = 1$, then the names of C_A and C_B have the same set of tokens.

Example 2: The *CamelS* similarity between two classes C_A=`HorizontalAxis` and C_B=`HorizontalCategoryAxis` is computed as follows. $T(C_A) = \{$Horizontal, Axis$\}$ and $T(C_B) = \{$Horizontal, Category, Axis$\}$ are the set of tokens in C_A and C_B names. $|T(C_A)| = 2$, $|T(C_B)| = 3$, $T(C_A) \cap T(C_B) = \{$Horizontal, Axis$\}$, $|T(C_A) \cap T(C_B)| = 2$. The *CamelS* similarity between C_A and C_B is:

$$CamelS(C_A, C_B) = \frac{2 \times 2}{2 + 3} = 0.8$$

The Levenshtein Edit Distance[83] between the names of C_A and C_B returns the number of edit operations (insertions, deletions, and substitutions) of characters required to transform the name of C_A into that of C_B. To have comparable Levenshtein distances, we use the normalised edit distance (ND), given by:

$$ND(C_A, C_B) = \frac{LEV(C_A, C_B)}{sum(length(C_A), length(C_B))} \in [0, 1]$$

where LEV computes the Levenshtein distance (we count substitution as an edit operation with cost 1, not as a deletion followed by an insertion with cost 2). If $ND(C_A, C_B) = 0$, then the names of C_A and C_B are the same. If $ND(C_A, C_B)$ is close to 1, then the names of C_A and C_B are different.

Example 3: Let C_A=`HorizontalAxis` and C_B=`HorizontalCategoryAxis`. The Normalised Edit Distance $ND(C_A, C_B) = 0.22$.

3.1.2.3 Combination of Similarities

We combine ND and $CamelS$ to compare the textual similarity between names of an original class C_A and some candidate renamed classes C_{B_i} $i \in [1, n]$, because ND and $CamelS$ assess different aspects of string comparison: ND is concerned with the difference between strings but cannot tell if they have something in common, while $CamelS$ focuses on their common tokens but cannot tell how different the other tokens are. Our algorithm reports the C_{B_j} $j \in [1, n]$, with the highest $CamelS$ and the lowest ND scores as the class renamed from C_A. If C_{B_j} has ND lower than the 0.40 threshold and $CamelS$ higher than the 0.50 threshold. Else, C_A is considered as deleted.

Algorithm 1 presents the pseudo-code of how we combine structural and textual similarities. When we compare the similarities of an original class C_A to many candidate renamed classes $\{C_{B_1}, ..., C_{B_n}\}$, we first compare their structural similarity $StrS$. We select the subset of candidate renamed classes having the highest $StrS$ value. Then, we compute their textual similarities using ND and $CamelS$. We select a best candidate renamed class C_{B_j} that has the lowest ND and the highest $CamelS$. Then, we compare its ND and $CamelS$ similarities to given thresholds. If C_{B_j} has not ND lower than the 0.40 threshold and $CamelS$ higher than the 0.50 threshold, we consider that class C_A was deleted and not renamed. Else, we consider that class C_A was renamed to C_{B_j}.

Previous authors[39] have fixed the threshold value of normalized edit distance (ND) to 0.40. We set the 0.5 threshold of $CamelS$ similarity through our experimental evaluations on two systems: JFreeChart and XercesJ (see Section 3.3).

Example 4: We want to identify a candidate renamed class that has the most similar name to the original class `DataSource` between JFreeChart v0.5.6 and v0.6.0. Let us assume that three candidate renamed classes `DataSet`, `Dataret`, and `DataSetdescription`, have the highest $StrS = 0.70$ score. Then, we compute their textual similarities (ND and $CamelS$). Both `DataSetdescription` and `DataSet` have the same $CamelS = 0.50$, while their ND is different (0.42 for `DataSetdescription` and 0.29 for `DataSet`). Also, `DataSet` and `Dataret` have the same $ND = 0.29$, while their $CamelS$ is different (0.0 for `Dataret` and 0.50 for `DataSet`). Thus, by combining ND and $CamelS$, we can identify that `DataSet` has the lowest ND and the highest $CamelS$. Then, by comparing the ND and $CamelS$ similarities to the given thresholds, we conclude that `DataSet` has ND lower than 0.40 threshold and $CamelS$ equal to 0.5 threshold. Thus, `DataSet` is the most similar to `DataSource`. We inspected the source code of JFreeChart (v0.5.6 and v0.6.0), our manual validation reveals that class `DataSource` was indeed re-

Algorithm 1 Similarities Combination Principle.

```
1: $R \leftarrow EmptyList\{\}$
2: $S \leftarrow EmptyList\{\}$
3: $camelThreshold = 0.50$
4: $ndThreshold = 0.40$
5: $A \leftarrow List\{\{C_{A_1}, ..., C_{A_n}\}$, candidate renamed classes $(version1)\}$
6: $B \leftarrow List\{\{C_{B_1}, ..., C_{B_m}\}$, candidate target classes $(version2)\}$
7: for each Class $C_{A_i}$ in $A$, $i \in [1, n]$ do
8:    for each Class $C_{B_j}$ in $B$, $j \in [1, m]$ do
9:       Compute Similarity $StrS(C_{A_i}, C_{B_j})$.
10:      if $StrS(C_{A_i}, C_{B_j}) > strMax$ then
11:         $R \leftarrow EmptyList\{\}$.
12:         ADD $C_{B_j}$ to $R$.
13:         $strMax \leftarrow StrS(C_{A_i}, C_{B_j})$.
14:      else
15:         if $StrS(C_{A_i}, C_{B_j}) = strMax$ then
16:            ADD $C_{B_j}$ to $R$.
17:         end if
18:      end if
19:   end for
20:   for each Class $C_{B_r}$ in $R$, $r \in [1, |R|]$ do
21:      Compute Similarity $CamelS(C_{A_i}, C_{B_r})$.
22:      Compute Similarity $ND(C_{A_i}, C_{B_r})$.
23:      $ndMin \leftarrow Min(ND(C_{A_i}, C_{B_r}), ndMin)$.
24:      $camelMax \leftarrow Max(CamelS(C_{A_i}, C_{B_r}), camelMax)$.
25:   end for
26:   for each Class $C_{B_r}$ in $R$, $r \in [1, |R|]$ do
27:      if $ND(C_{A_i}, C_{B_r}) == ndMin$ AND $CamelS(C_{A_i}, C_{B_r}) == camelMax$ then
28:         if $ND(C_{A_i}, C_{B_r}) <= ndThreshold$ AND $CamelS(C_{A_i}, C_{B_r}) >=$
            $camelThreshold$ then
29:            $S \leftarrow \{C_{B_j}\}$, having $ndMin$ and $camelMax$
30:         end if
31:      end if
32:   end for
33:   if $|S| = 0$ then
34:      Class $C_{A_i}$ is deleted.
35:   else
36:      Class $C_{A_i}$ is renamed to $C_{B_j}$.
37:   end if
38: end for
```

named to `DataSet`.

Example 5: we want to identify a candidate renamed class that has the most similar name to the original class `BlankAxis` between JFreeChart $v0.5.6$ and $v0.6.0$. `HorizontalDateAxis` and `HorizontalCategoryAxis` are two candidate classes having the highest $StrS$ scores (0.66) to `BlankAxis`. Then, we compute their textual similarities (ND and $CamelS$).

Both `HorizontalDateAxis` and `HorizontalCategoryAxis` have the same $CamelS$ equal to 0.40. However, it is not higher than the 0.50 threshold. Also, their ND similarities are not lower than the 0.40 threshold (0.44 for `HorizontalDateAxis` and 0.51 for `HorizontalCategoryAxis`). Thus, by comparing the ND and $CamelS$ similarities to the given thresholds, we conclude that the original class `BlankAxis` was deleted and not renamed. We inspected the source code of JFreeChart (v0.5.6 and v0.6.0), our manual validation reveals that class `BlankAxis` was indeed deleted.

3.1.3 Step 3: Design Diagram Matching

Given two subsequent design versions and the names of renamed classes, we now use a bit-vector algorithm [13] to match the two design versions to each other. We summarise our iterative bit-vector algorithm for software design matching as follows: we first convert software designs into strings, because bit-vector algorithms are designed for strings, defined by the sequence of triplets $T = (C_{Source}, R, C_{Target})$, each triplet representing a relation between the two classes C_{Source} and C_{Target}. Then, we analyse these strings to identify the sets of stable and deleted triplets using a bit-vector algorithm. This algorithm consists of traversing the string representation of the first version, triplet by triplet, then recording the triplets in the first version that match those in the second version. Finally, we obtain the sets of stable triplets that exist in all versions and the set of deleted triplets. We use these sets to find stable ($S\mu_D$) and decaying ($D\mu_D$) micro-designs, which we use to measure the impact of design decay, such as bug proneness and design defect proneness.

3.1.3.1 String Representations of Software Designs

We use an existing tool, EPI [66], to convert the software designs [50] previously generated by PADL, into string representations, defined by sequences of triplets $T = (C_{Source}, R, C_{Target})$, each triplet representing a connection between the C_{Source} and the C_{Target}. This conversion consists of two steps:

- First, it takes as input the digraph (software design) previously generated by PADL. Then, it transforms the digraph into a Eulerian digraph (see Figure 3.2(b)). A digraph is typically not Eulerian, because it does not contain a Eulerian circuit, *i.e.*, a path that passes through each edge exactly once. A digraph is Eulerian if and only if every vertex has equal numbers of incoming and outgoing edges for a vertex. The transformation consists in adding "dummy" edges, noted *dm*, between vertices with unequal numbers of incoming and outgoing edges. EPI uses the transportation simplex [62] to obtain the number of dummy edges to be added.
- Second, by traversing the minimum Eulerian circuit, it generates a unique string representation of the software design (see Figure 3.2(c)). Hence, we it solves the directed Chinese Postman problem: the shortest path of a graph that visits each edge at least once [38].

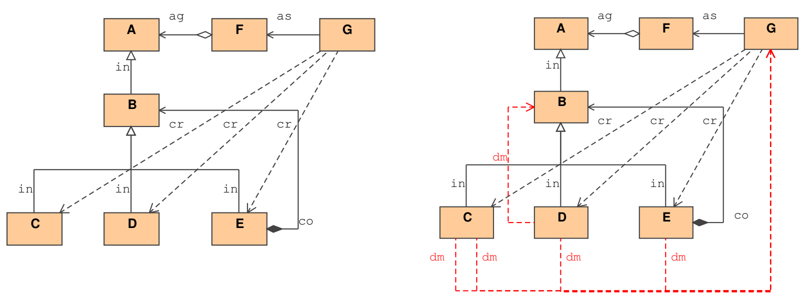
3.1.3.2 Characteristic Vectors

To use a bit-vector algorithm for matching two subsequent versions of software designs, we use the string representation $S = t_1 ... t_m$, as $V = (v_1 ... v_m)$ of version 2 and the set ST of all tokens in the string representation of version 1 as input, then, for each token t in ST, we build the characteristic vector of the token t associated with the string $S = t_1 ... t_m$, as $V = (v_1 ... v_m)$:

$$v_i = \begin{cases} 1 & \text{if } t_i = t \\ 0 & \text{otherwise.} \end{cases}$$

Example 6: For the example shown in Figure 3.2(c), the characteristic vectors of tokens A, in, and B are defined as:

(a) UML-like class diagram (b) Eulerian model



(c) String representation of the Eulerian model

```
A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A
```

Figure 3.3 – String Representation of a software design (from [66]).

$$
\begin{aligned}
A &= \mathbf{1}\underbrace{0000000000000000}_{30}\mathbf{1} \\
in &= 0\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{0000000}_{19} \\
B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000}_{20}
\end{aligned}
$$

Characteristic vectors are sequences of bits on which we operate with: bit-wise logical AND ($\wedge$), OR ($\vee$) operators, left ($\leftarrow$) and right ($\rightarrow$) shifts. We define the right shift of a characteristic vector $V = (v_1, v_2..., v_{m-1}, v_m)$ by shifting all the bits to the right by one position (circularly) as $\rightarrow V = (v_m, v_1, v_2..., v_{m-1})$. Similarly, the left shift of v is $\leftarrow V = (v_2..., v_{m-1}, v_m, v_1)$.

3.1.3.3 Bit-vector Algorithm

Given the string representation of system version 1, the characteristic vectors of system version 2, and the names of renamed classes, we now use a bit-vector algorithm to match the designs of two system versions to each other. It can find all stable triplets between two versions of a design in a bounded number of vector operations, regardless of the length of the input (*i.e.*, the number of tokens of each string representation). Such an algorithm can be implemented with bit-wise operations available in processors, leading to highly efficient computations [13].

Algorithm 2 works as follows: Let $SetTriplets$ be the set of all triplets of the first version, $Vectors$ be the set of all characteristic vectors of the second version, and $Renamings$ be the set of the names of renamed classes. First, our algorithm traverses the set $SetTriplets$, then for each triplet $T = (C_{Source}, R, C_{Target})$ in $SetTriplets$: If $Vectors$ contains the characteristic vector of C_{Source} and C_{Target} then we initialize $vectorSource$ with the characteristic vector of C_{Source}, and the $vectorTarget$ with the characteristic vector of C_{Target}, and $vectorR$ with the characteristic vector of R. Then, we apply bit-wise operations on those characteristic vectors to compute the conjunction : $(\rightarrow\rightarrow vectorSource) \wedge (\rightarrow vectorR) \wedge (vectorTarget)$. If the conjunction is not $NULL$, then T is added to the set $StableTriplets$. Else, T is considered deleted and added to the set $DeletedTriplets$. If C_{Source} or C_{Target} were renamed, then we use the characteristic vector of the new class name, of C_{Source} or C_{Target}, using the list of class renamings (see Step 2). If C_{Source} or C_{Target} were deleted in the second version (as described in Step 2), then the triplet T is considered deleted and is added to the set $DeletedTriplets$. Finally, our algorithm stores the triplets identified as being stable or deleted between two design diagram versions.

Example 7: Let assume that the triplet $T = (A, in, B)$ exists in the first version, and we would like to verify whether T exists in the second version too. Thus, we build characteristic vectors of each token of T from the string representation of the second version to compute the following conjunction:

Algorithm 2 Bit-Vector Principle.

```
1: $StableTriplets \leftarrow EmptyList$ {}
2: $DeletedTriplets \leftarrow EmptyList$ {}
3: $Renamings \leftarrow List$ { All renamings (sourceClass , renamedClass)}
4: $SetTriplets \leftarrow List$ {All triplets of system version 1}
5: $Vectors \leftarrow List${All characteristic vectors of system version 2}
6: for each Triplet $T = (C_{Source}, R, C_{Target})$ in $SetTriplets$ do
7:    $vectorR = CharacteristicVector(R)$
8:    if $Vectors$ contains $CharacteristicVector(C_{Source})$ then
9:       $vectorSource = CharacteristicVector(C_{Source})$
10:   else
11:      if $Renamings$ contains $C_{Source}$ then
12:         $Renamed_{Source} = getRenamings(C_{Source})$
13:         $vectorSource = CharacteristicVector(Renamed_{Source})$
14:      else
15:         ADD $T$ to $DeletedTriplets$
16:         continue
17:      end if
18:   end if
19:   if $Vectors$ contains $CharacteristicVector(C_{Target})$ then
20:      $vectorTarget = CharacteristicVector(C_{Target})$
21:   else
22:      if $Renamings$ contains $C_{Target}$ then
23:         $Renamed_{Target} = getRenamings(C_{Target})$
24:         $vectorTarget = CharacteristicVector(Renamed_{Target})$
25:      else
26:         ADD $T$ to $DeletedTriplets$
27:         continue
28:      end if
29:   end if
30:   $Conjunction = (\rightarrow\rightarrow vectorSource) \wedge (\rightarrow vectorR) \wedge (vectorTarget)$
31:   if $Conjunction$ is not $NULL$ then
32:      ADD $T$ to $StableTriplets.$
33:   else
34:      ADD $T$ to $DeletedTriplets.$
35:   end if
36: end for
```

$$\begin{aligned}
(\rightarrow\rightarrow A) &= 0\mathbf{11}\,\underbrace{0000000000000000000}_{29} \\
(\rightarrow in) &= 00\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\,\underbrace{000000}_{18} \\
B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\,\underbrace{00000000}_{20} \\
Conjunction &= (\rightarrow\rightarrow \mathbf{A}) \wedge (\rightarrow \mathbf{in}) \wedge B \\
&= 00\mathbf{1}\,\underbrace{00000000000000000000}_{29}
\end{aligned}$$

and assess whether the bit vector $Conjunction$ is null (contains only zeros). If the $Conjunction$ is not null, then the class B is related to class A through the inheritance relation in (in the second version) and thus, the triplet $T = (A, in, B)$ is stable between the two versions.

3.1.4 Step 4: Design Diagram Clustering

Once we obtained the set of all stable and deleted triplets between two design diagram versions using the bit-vector algorithm, we apply our incremental clustering algorithm to find the sets of connected triplets that form the sets of stable ($S\mu_D$) and decaying ($D\mu_D$) micro-designs between two design versions.

To find stable micro-designs ($S\mu_D$), our incremental clustering algorithm requires one and only one scan of all stable triplets. Each triplet is read and then either assigned to one of the $S\mu_D$s or used to start a new $S\mu_D$. Then, the set of existing $S\mu_D$s is reduced by merging two $S\mu_D$s if a new triplet join them, *i.e.*, it includes a relation between classes belonging to the two $S\mu_Ds$.

We describe our Algorithm 3 as follows: Let S be the list of all stable triplets. First, it traverses S, then for each triplet T in S and for each cluster C, if there is a triplet T^* in C that has a relation with T, then the triplet T is added to the cluster C, which is also marked as *Cluster to be merged* (lines 3-8). If there is another cluster that contains another triplet T^* that has a relation with triplet T, then the current cluster is merged with the marked cluster (lines 9-10). If, after

Algorithm 3 Incremental Clustering Principle.

```
1: $L \leftarrow EmptyList\{Clusters\}$
2: $S \leftarrow List$ {Stable Triplets between two system versions}
3: for each Triplet $T$ in $S$ do
4:   for each Cluster $C$ in $L$ do
5:     if $T$ has a relation with the existing triplet $T^*$ in $C$ then
6:       if $T$ is not added to any cluster then
7:         ADD $T$ to $C$.
8:         $ClusterToBeMerged \leftarrow C$.
9:       else
10:        MERGE $ClusterToBeMerged$ to $C$.
11:      end if
12:    end if
13:  end for
14:  if $T$ is not added to any cluster then
15:    Create a new Cluster $C^*$.
16:    ADD $T$ to $C^*$.
17:    ADD $C^*$ to $L$.
18:  end if
19: end for
```

checking all clusters C in L, the triplet T was not assigned to any cluster, a new cluster C^* is created and the triplet T is added to it (lines 14-17).

Our algorithm returns the clusters that represent stable micro-designs $S\mu_D$s between two versions. Similarly, we can apply the same algorithm to find decaying micro-designs ($D\mu_D$) using the set of deleted triplets as input.

3.1.5 Step 5: Design Decay Evaluation

Given the set of stable micro-designs of each subsequent pair of software designs, we can measure the design stability, as follows:

- *Tunnel Triplets Metric (TTM)*: this metric represents the stability of the design with respect to the original design (the first version). It reports the number of triplets that have a match in all the versions. These triplets are considered to be the backbone of the system, *i.e.*, a kind of tunnel across time. Let $S_{Tunnel(T)}$ be the set of triplets $T = (C_{Source}, R, C_{Target})$ that are present in the $S\mu_D$ from the first to the last version of a software design. We define, $TTM(Version_i)$, the number of tunnel triplets at $Version_i$ as:

$$S_{Tunnel(T)} = \{T | T \in \{S\mu_D(j) \bigcap S\mu_D(j+1), \forall j \in [0,i]\}\}$$

$$S\mu_D(Version_i) = \{T_k | T_k \in Version_j, \forall k \in [0,N], \forall j \in [0,i]\}$$

$$TTM(Version_i) = |S_{Tunnel(T)}|$$

- *Common Triplets Metric (CTM)*: this metric represents the stability of the design with respect to the previous version. It consists of computing the number of triplets that have not changed since their first appearance in a given version. Let $S(T)$ be the set of triplets $T = (C_{Source}, R, C_{Target})$ that are never deleted since their first appearance. We define, $CTM(Version_i)$, the number of stable triplets at $Version_i$ as:

$$S(T) = \{T | T \in V_j, \forall j \in [k,i], \exists k \in [0,i[\}$$

$$CTM(Version_i) = |S(T)|$$

We show in Section 3.2 how these measures provide useful insights to developers regarding the evaluation of design decay in object-oriented systems.

3.2 Study Definition and Design

Following the Goal Question Metric (GQM) methodology [10], the *goal* of our study is to analyse the performance of our approach $ADvISE$. The *purpose* is to provide an approach for identifying class renaming and evaluating design decay. The *quality focus* is to evaluate the design decay of software systems, and to provide a set of renamings occurrences with good precision and recall and in a reasonable time. The *perspective* is that of both researchers who want to study class renaming, and practitioners who analyse software evolution to estimate the effort required for future maintenance tasks. The *context* of our experiments is five open-source Java systems: ArgoUML, DNSjava, JFreeChart, Rhino and XercesJ.

3.2.1 Objects

We perform our study on five well-known, open-source software systems: ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ. We selected these systems because: (1) they are open source belonging to different domains, (2) several versions of these systems are available, (3) the lengths of their histories are long enough to make interesting observations on the signs of the design decay, (4) they vary from medium-sized to large open-source projects, (5) defect data (bugs and design defects) are available from previous authors [36, 71] for Rhino and ArgoUML, (6) these systems were previously studied in previous work [36, 71, 76, 77]. The last condition reduces the bias in the selection of the subject systems and facilitates the comparison with previous work. Table 3.1 shows some descriptive statistics of these systems.

ArgoUML[3] is a graphical software design environment that supports the design, development and documentation of object-oriented software systems. DNSjava[4] is an implementation of the DNS protocol in Java. DNSjava includes a set of classes that can be used within other systems and several user tools. JFreeChart[5] is a powerful and flexible open-source charting library. Rhino[6] is an open-source implementation of JavaScript written entirely in Java. XercesJ[7] is a family of software packages for parsing XML.

3.2.2 Research Questions

We break down our study into four phases. First, we study the performance of our class renaming detection and we seek answers to the following questions:

- **RQ1: What are the thresholds for class renaming detection?** This question aims at studying how thresholds can be systematically derived for our renaming detection technique.

[3]`http://argouml-stats.tigris.org/`
[4]`http://www.dnsjava.org/`
[5]`http://www.jfree.org/jfreechart/`
[6]`http://www.mozilla.org/rhino/`
[7]`http://xerces.apache.org/`

System	Releases	Entities (in classes)	Bit-vectors (in bits)	History (in releases)	Dates
ArgoUML	From v0.10.1	1447	12,265,560	17	07/07/2002
	To v0.34	1984	105,456,260		15/12/2011
DNSjava	From v1.2.0	164	49,759	33	07/04/2001
	To v2.1.3	124	93,067		24/10/2011
JFreeChart	From v0.5.6	100	87,227	51	25/11/2000
	To v1.0.13	778	1,089,345		20/04/2009
Rhino	From v1.5.R1	163	40,803	11	10/05/1999
	To v1.6.R5	449	266,265		19/11/2006
XercesJ	From v1.0.	296	162,583	36	05/11/1999
	To v2.9.0	697	1,195,353		22/11/2006

Table 3.1 – Statistics for the first and last version of each system.

- **RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?** This question aims at studying the performance of $ADvISE$ in terms of precision and recall for class renaming detection. We investigate how our structure-based and text-based similarities can help the identification of class renaming in the evolution of a software system.

Then, we investigate whether it is possible to apply our approach to study the design decay of object-oriented software systems:

- **RQ3: What are signs of design decay and how can they be tracked down?** This question aims at studying whether TTM and CTM are good indicators of design decay and if they provide useful insights to developers regarding the signs of software aging.

Then, we investigate whether decayed designs make systems more prone to bugs and design defects:

- **RQ4: Do stable and decaying micro-designs have the same risk to be bug-prone?** This question leads to the following null hypothesis:
  - H_0**:** There is no significant difference between the proportions of bugs carried by stable ($S\mu_D$) and decaying micro-designs ($D\mu_D$).
- **RQ5: Are decaying micro-designs more prone to design defects than stable micro-designs?** This question leads to the following null hypothesis:

  - H_0**:** There is no significant difference between the proportions of design defects carried by stable ($S\mu_D$) and decaying micro-designs ($D\mu_D$).

Finally, we study the performance of our approach $ADvISE$:

- **RQ6: How does ADvISE perform?** This question aims at studying the performance of $ADvISE$ outlining the execution time for each step and for each examined system.

3.2.3 Analysis Methods

We perform the following analyses to answer the research questions:

RQ1: What are the thresholds for class renaming detection?
For **RQ1**, we compute the F-measure of the class renamings for different threshold values, in comparison to our oracle (see RQ2) to find the optimal threshold values of the ND and $CamelS$ similarities for JFreeChart and XercesJ. The maximum values of the F-measure correspond to a balanced compromise between precision and recall [8], threshold values before the peak favor precision, while threshold values after the peak promote recall.

$$F - measure = \frac{2 \times (precision \times recall)}{(precision + recall)}$$

$$precision = \frac{|correct \cap detected|}{|detected|}$$

$$recall = \frac{|correct \cap detected|}{|correct|}$$

where *correct* represents the set of known renamed classes and *detected* that of candidate occurrences detected by our approach.

RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?
For **RQ2**, we first apply our approach on JFreeChart and XercesJ to detect class renamings using one type of similarity at a time, in order to investigate how our

structural similarity ($StrS$), textual similarities ($CamelS$ and ND), and their combination can help the identification of class renaming in a software system.

We need an oracle to study the efficiency of ADvISE and validate its detection of class renamings. Such an oracle must provide for a set of systems, the true class renamings between different pairs of versions. Manually building the oracle for all releases is a time consuming task even for small sized systems, such as XercesJ. Producing an oracle for medium sized systems may require inspection of thousands of matches. Indeed, there are about $3,570$ class renaming candidates for 51 releases of JFreeChart. Thus, an exhaustive manual verification is not feasible.

To reduce the required manual verification, we propose a semi-automated oracle building process that consists of three main phases:

- First, we apply ADvISE on JFreeChart and XercesJ to detect class renamings.
- Second, we use a *stratified random sampling* to gather a representative set of class renamings for each system.
- Finally, we applied the ECGM algorithm [75] on the set of sampled Java classes. Let $S(ADvISE)$ and $S(ECGM)$ be the set of class renamings detected by $ADvISE$ (respectively, $ECGM$). We consider that $S(ADvISE) \cap S(ECGM)$ is a true class renaming and we inspect manually the differences $S(ADvISE) - S(ECGM)$ and $S(ECGM) - S(ADvISE)$ to build our oracle.

Stratified Random Sampling is a probability sampling technique, also sometimes called proportional random sampling. It is used to estimate population parameters efficiently when there is substantial variability between sub-populations [33]. A stratum is a subset of elements in the population sharing at least one common characteristic, *i.e.*, they have similar values of one or more stratification variables. The values of the stratification variables are known for the entire population. This technique involves partitioning the entire population (N) into homogeneous subgroups (S_1, S_2, S_3,..., S_i) called *strata*, such that $N = S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$, and then taking a random sample proportional to the fraction n_i/N in each subgroup S_i, where n_i is the sample size of S_i. With the stratified sampling technique, we have a higher statistical precision compared to simple random sampling, because the variability within the subgroups is lower compared to the variations when dealing with the entire population.

In our case, we divide the set of fully qualified class names of all system versions into two subgroups (*strata*) corresponding to two types of renamings:

1. Class renaming with or without changing the package name;
2. Package renaming without changing the class name.

Then, we randomly select from each subgroup as explained above. We compute the total sample size using the following formula:

$$Sample_{size} = \frac{Z^2 \times P \times (1 - P)}{C^2}$$

Z= Z value, a standard value of 1.96 (the confidence level at 95%).

P = Percentage of picking a choice (0.5 used for sample size needed).

C = Confidence Interval, a standard value of 0.05 (the margin of error at 5%).

Table 3.2 shows the amount of class renamings in JFreeChart and XercesJ. There are 3,570 cases of class renamings for 51 releases of JFreeChart (respectively, 715 for 36 releases of XercesJ). Table 3.3 reports the results of the stratified sampling. The sample size of JFreeChart is 347 (respectively, 250 for XercesJ). In Table 3.3, we divided the set of class renamings into two *strata* using the percentage of each subgroup obtained from Table 3.2.

System	**Releases**	**Population Size**	**# Package Renamings**	**# Class Renamings**
JFreeChart	v0.5.6-v1.0.14	3,570 (100%)	2,127 (59.57%)	1,443 (40.43%)
XercesJ	v1.0.1-v2.0.9	715 (100%)	389 (54.40%)	326 (45.60%)

Table 3.2 – Amount of class renaming.

Once we select the stratified random sample, we collect the numbers of true and false positive occurrences of the class renamings in the sample using our oracle, using the precision and recall values [8]:

$$precision = \frac{|correct \cap detected|}{|detected|}$$

System	Releases	Sample Size	Stratum 1 Size	Stratum 2 Size
JFreeChart	v0.5.6-v1.0.14	347 (100%)	207 (59.57%)	140 (40.43%)
XercesJ	v1.0.1-v2.0.9	250 (100%)	136 (54.40%)	114 (45.60%)

Table 3.3 – Sample sizes after startified Sampling.

$$recall = \frac{|correct \cap detected|}{|correct|}$$

where *correct* represents the set of known renamed classes and *detected* that of candidate occurrences detected by our approach.

RQ3: What are signs of design decay and how can they be tracked down?
For **RQ3**, we first apply our approach to ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ. Then, we perform a pair by pair matching of subsequent software designs to identify stable triplets in these systems. To evaluate the deviation of the actual design from the original design, we compute the number of triplets that has a match in all the versions, using our TTM metric. These triplets are considered to be part of a tunnel, *i.e.*, the backbone part of the system. Also, to analyse the stability of the design with an enriched functionality, we compute the number of triplets that have not changed since their first appearance in a given version of a system, using our CTM metric. Then, we build a graph visualising the evolution of a software design over time. The axes of the graphic are: the time (software versions) and the values of our indicators of design decay (number of triplets in the tunnel (TTM) and number of common triplets between two versions (CTM)). Then, we validate the graph of designs evolution for each system using external information provided by bug reports, mailing lists, and release notes to assess whether these indicators provide useful insights regarding the signs of software aging.

RQ4: Do stable and decaying micro-designs have the same risk to be bug-prone?
For **RQ4**, we first apply the bit-vector algorithm to identify stable ($S\mu_D$) and

decaying ($D\mu_D$) micro-designs in ArgoUML and Rhino, using publicly available data on the bugs collected [8] by previous authors [36, 71].

To test H_0, we test whether the proportion of classes that compose decaying (respectively stable) micro-designs take part (or not) in significantly more faults than those in stable (respectively decaying) micro-designs. We use contingency tables to assess the direction of the difference, if any. We use Fisher's exact test [113], to check whether the difference is significant. We also compute the odds ratio [113] that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds that decayed classes are identified as fault-prone, to the odds that stable classes are identified as fault-prone. An odds ratio greater than 1 indicates that the event is more likely in the first sample (decayed classes), while an odds ratio less than 1 indicates that it is more likely in the second sample. An odds ratio $OR = \frac{p}{(1-p)}$. We expect $OR > 1$ and a statistically significant p-value.

RQ5: Are decaying micro-designs more prone to design defects than stable micro-designs?

For **RQ5**, we first apply the bit-vector algorithm to identify stable micro-designs in ArgoUML and Rhino, using publicly available data on the design defects collected[9] by previous authors [71]. In this study, we consider 12 types of design defects, such as: Anti-Singleton, Blob, ClassDataShouldBePrivate, ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList, MessageChains, RefusedParentRequest, SpeculativeGenerality, and SwissArmyKnife (see Appendix B).

To attempt rejecting H_0, we test whether the proportion of classes that compose decaying (respectively stable) micro-designs take part (or not) in significantly more design defects than those in stable (respectively decaying) micro-designs. We use contingency tables to assess the direction of the difference, if any. We use Fisher's exact test [113], to check whether the difference is significant. We also compute the odds ratio that indicates the likelihood for an event to occur. We expect $OR > 1$ and a statistically significant p-value.

[8]ArgoUML `http://www.ptidej.net/downloads/experiments/emse10`
Rhino `http://www.cs.columbia.edu/~eaddy/concerntagger/`

[9]`http://www.ptidej.net/downloads/experiments/emse10`

RQ6: How does ADvISE perform?
For **RQ6**, we first apply our approach on ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ, to investigate the impact of the size of software designs on matching time. For each system, we apply our approach ADvISE to pairs of releases. Then, we record the median across multiple runs for a particular configuration. ADvISE'algorithms are coded in Java and run using a standard Intel Core i5 (2.53GHz) with 6GB RAM running Microsoft Windows 7 (64-bit). We measure the time performance in Java using *System.currentTimeMillis()*. This method returns the current time in milliseconds since midnight GMT on January 1st, 1970. We use this technique, because we are interested in profiling events that run measurably slow (more than 1 millisecond).

3.3 Empirical Study Results

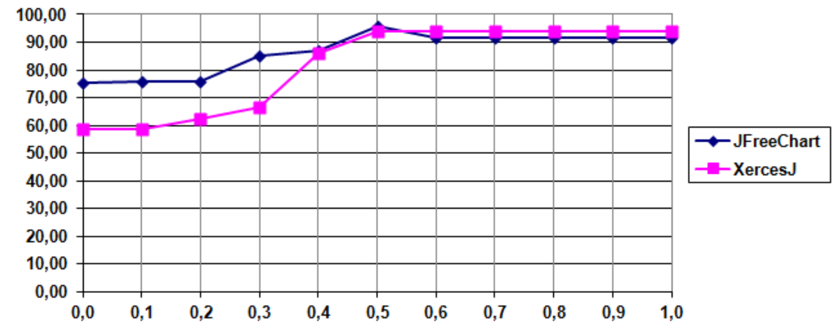
We now report and discuss the results of our study.

RQ1: What are the thresholds for class renaming detection?
Figure 3.4 shows the $F - measure$ graph of class renamings in JFreeChart and XercesJ with different threshold values. For $CamelS$, we select 0.5 as best threshold value, in which the optimal $F - measure$ values are 95.99 for JFreeChart and 93.96 for XercesJ.

For ND, we have a range of threshold values $Range = [0.4, 0.7]$, in which the optimal $F - measure$ values are 95.86 for JFreeChart and 85.95 for XercesJ. Previous authors[39] have fixed the threshold value of normalized edit distance (ND) to 0.40, which belongs to our Range. Thus, we select 0.4 as best ND threshold.

RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?
In Table 3.4, we present the precision and recall on each subject system. The first three columns represent the efficiency of class renaming using just one similarity value ($CamelS$, ND, $StrS$). $CamelS$ similarity alone provides the best results for XercesJ with a precision of 84.61%. However, it provides the worst results for JFreeChart with a precision of 65.90%. ND similarity alone provides the best

(a) Camel Similarity



(b) Normalized Edit Distance

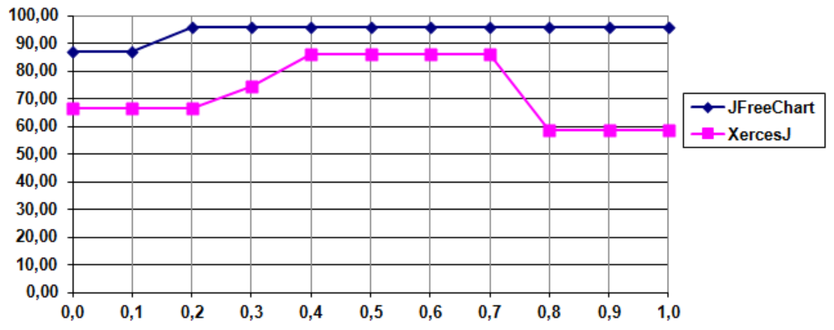


Figure 3.4 – F-measure in function of thresholds values for *CamelS and ND Similarities.*

results for JFreeChart with a precision of 77.27%, but it provides the worst results for XercesJ with a precision of 38.46%. The precision of *StrS* similarity is 72.72% for JFreeChart and 57.69% for XercesJ.

In the last column, we present the efficiency of our approach using the combination of all similarities. This combination yields a precision of 95.45% in JFreeChart (respectively, 92.30% in XercesJ), while the recall of JFreeChart is 97.67% (respectively, 96.00% in XercesJ). We conclude that the combination of structural and textual similarities provide better results than each single similarity.

Systems	**Similarities**	**CamelS**	**ND**	**StrS**	**Combination**
JFreeChart v0.5.6-v1.0.13	Precision	65.90%	77.27%	72.72%	95.45%
	Recall	67.41%	79.06%	74.41%	97.67%
XercesJ v1.0.1-v2.9.0	Precision	84.61%	38.46%	57.69%	92.30%
	Recall	88.00%	40.00%	60.00%	96.00%

Table 3.4 – The performance of *ADvISE* in terms of precision and recall for class renaming detection.

RQ3: What are signs of design decay and how can they be tracked down? Our bit-vector and incremental clustering algorithms identified the common triplets in ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ. We then compared, on the one hand the graphs visualising the number of triplets between two subsequent versions, and on the other hand, external information in the bug reports, release notes and mailing lists.

ArgoUML

The number of triplets in the tunnel (TTM metric) was decreasing from 10,140 to 7,398 triplets at version 0.26 (see Figure 3.5). Then, it remains stable throughout the life of ArgoUML, which means that the design of argoUML evolves without affecting its original design. Similarly, the number of common triplets (CTM metric) between versions 0.24 and 0.26 decreased from 11,234 to 8,363 triplets. We inspected the set of deleted triplets, and found a large decaying micro-design containing 1879 triplets. Using external information[10], [11], we explain the results shown in Figure 3.5 as follows:

- **0.24 – 0.26**: CTM metric between versions 0.24 and 0.26 decreased from 11,234 to 8,363 triplets. The release notes report an important remove activity of "*Group and Ungroup Actions*" and *GEF library* (gef-0.12.jar) on 30-08-2006. The release schedule reports that 0.24 is the last release supporting Java 1.4.
- **0.26 – 0.26.2**: CTM metric between versions 0.26 and 0.26.2 increased from 8,363 to 10,643 triplets. The release schedule reports that new features were

[10]http://argouml.tigris.org/wiki/ReleaseSchedule
[11]http://argouml.tigris.org/wiki/ReleaseSchedule/Past_Releases_in_Detail

added, such as explorer drag and drop, profiles, settable diagram fonts, and activity diagram swimlanes (partitions), etc.

- **0.26.2 – 0.28**: CTM metric between versions 0.26.2 and 0.28 decreased from 10,643 to 10,146 triplets. The release schedule reports critical bug fixes for multinational character support and profiles.

- **0.28 – 0.28.1**: CTM metric between versions 0.28 and 0.28.1 increased from 10,146 to 11,163 triplets. The release schedule reports the integration of the results of the GSoC[12] projects. A new sequence diagram implementation is provided. Additionally, functionalities is moved into separate modules, such as draggable edge labels, C# source import, new diagram icons.

- **0.28.1 – 0.30**: CTM metric between versions 0.28.1 and 0.30 decreased from 11,163 to 10,694 triplets. The release schedule reports the introduction of a new implementation of property panels in XML files.

- **0.30 – 0.30.1**: CTM metric between versions 0.30 and 0.30.1 increased from 10,694 to 11,215 triplets. The release schedule reports some transformations and bug fixes.

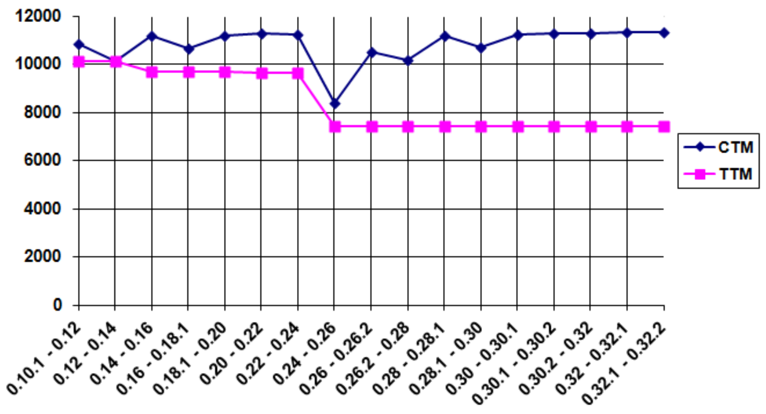


Figure 3.5 – The evolution of the ArgoUML design.

[12]Google Summer of Code Project http://argouml.tigris.org/wiki/Ideas_for_GSoC_2010

DNSjava

The number of triplets in the tunnel (TTM metric) decreased from 749 to 329 triplets throughout the life time of DNSjava (see Figure 3.6). The number of common triplets (CTM metric) between versions 1.5.2 and 1.6.1 decreased to 853 triplets. Using external information, we show that this period[13] (24-02-2004 and 16-03-2004) corresponds to the largest number of changes during two successive months[14].

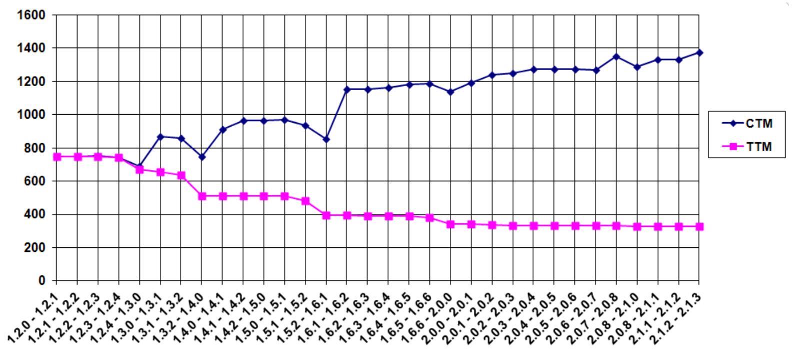


Figure 3.6 – The evolution of the DNSjava design.

JFreeChart

The number of triplets in the first (TTM metric) version decreased from 413 to 100 stable triplets in the tunnel (see Figure 3.7). This decrease is due to major changes that have been made. Using external information, we explain the results shown in Figure 3.7 as follows:

- **0.9.20 – 0.9.21**: The number of common triplets (CTM metric) between versions 0.9.20 and 0.9.21 decreased to 1,894 triplets, the release notes report an important splitting activity of two packages `org.jfree.data` and `org.jfree.chart.renderer` into sub-packages `category` and `xycharts`. The fully qualified names of all entities in both packages have been changed, which decreased the number of common triplets.

[13] `http://www.dnsjava.org/download/old/`

[14] `http://sourceforge.net/mailarchive/forum.php?forum_name=dnsjava-changes&max_rows=25&style=ultimate&viewmonth=200402`

- **0.9.21 – 1.0.0**: The CTM metric between versions 0.9.21 and 1.0.0 increased to $3,356$. The release notes report v1.0.0 to be the first stable release of the JFreeChart class library, all future releases in the 1.0.x series will aim to maintain backward compatibility with this release.
- **1.0.12 – 1.0.13**: The CTM metric between versions 1.0.12 and 1.0.13 increased again. After version 1.0.0, we noticed that the number of common triplets was increasing until the last version (1.0.13). The release notes reveal that some new features were added and some bugs fixed. The number of common triplets in the tunnel (TTM metric) remains constant, the backbone of the system is more stable.

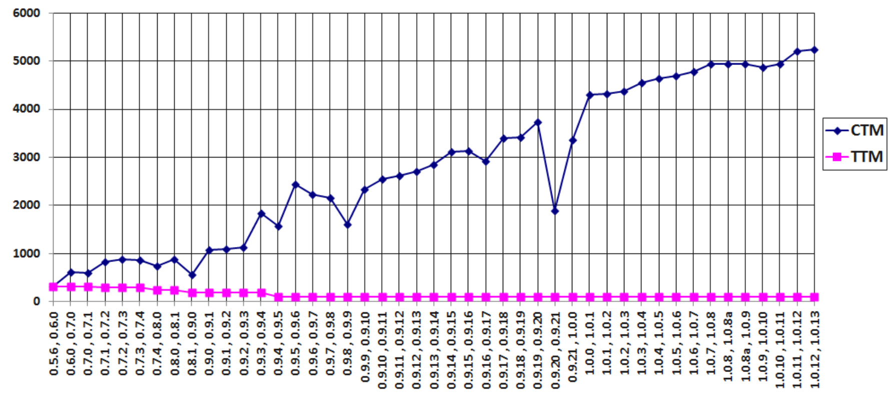


Figure 3.7 – The evolution of the JFreeChart design.

Our approach has the potential to discover two cases of renamings in a fully-qualified class name: (1) Class renaming without changing the package name; (2) Package renaming without changing the class name. In the second case, the triplets are not considered stable, because renamings are due to structural changes in the design, such as splitting or merging packages, moving the class to a new package.

Rhino

The number of triplets in the tunnel (TTM metric) was decreasing from 677 to 421 triplets throughout all the life of Rhino (see Figure 3.8). The number of common triplets (CTM metric) between versions 1.5R5 and 1.6R1 decreased from 1,335 to 1,228 triplets, while the TTM metric remains stable, the release notes report *Rhino 1.6R1 as the new major release of Rhino*, there are important changes in Rhino 1.6R1, *"... without affecting the existing code base"*[15]. Thus, the triplets in the tunnel represent the existing code base.

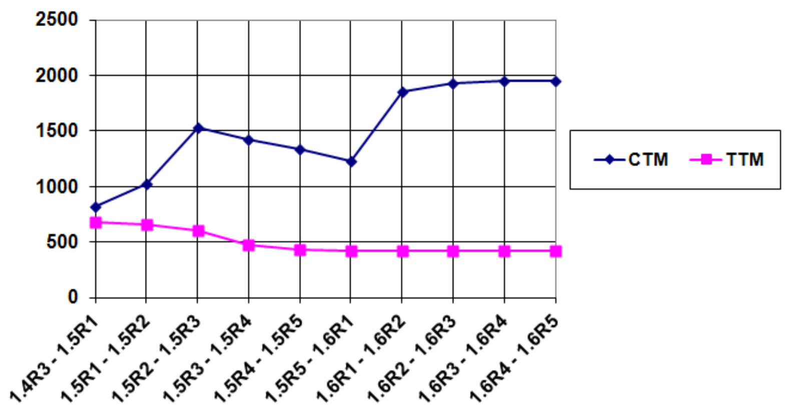


Figure 3.8 – The evolution of the Rhino design.

XercesJ

The number of triplets in the first version (TTM metric) decreased from 1,693 to 484 triplets throughout the life time of XercesJ (see Figure 3.9). This means that 28.58% of the triplets belong to the tunnel. Using external information, we explain the results shown in Figure 3.9 as follows:

- **1.0.1 – 1.4.4**: the number of common triplets (CTM metric) increased from 1,693 to 3,160 triplets, because new features were added and maintained until version 1.44. The number of stable triplets in the tunnel (TTM metric) decreased, some classes in the first version were deleted and replaced by new ones.

- **1.4.4 – 2.0.0**: the number of common triplets (CTM metric) decreased from 3,160 to 959 triplets. There were major changes reported in version 2.0.0.

[15]`http://www.mozilla.org/rhino/rhino16R1.html`

Also, the number of stable triplets in the tunnel (TTM metric) decreased from 1,693 to 488 triplets, because classes from the first version were deleted. The release notes report that "XercesJ 2.0.0 is a nearly complete rewrite of the XercesJ 1.x code base to make the code cleaner, more modular, and easier to maintain. It includes a completely redesigned and rewritten XML Schema validation engine".

- **2.0.0 – 2.0.9**: the number of common triplets increased from 959 triplets to 6,096. The software design became more stable, there were just new features added and some bugs fixed. Also, the number of stable triplets in the tunnel remained constant at 488, so the backbone of the system is now stable.

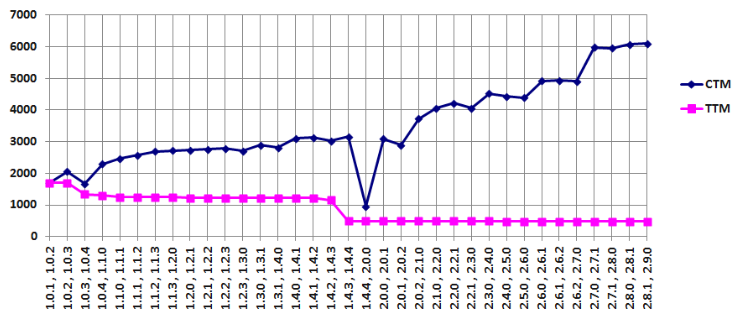


Figure 3.9 – The evolution of the XercesJ design.

Discussion

In our approach, a software design is represented by a (possibly reverse-engineered) class diagram. In ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ, the number of common triplets between two subsequent versions (CTM) is increasing over time. The first measure of design decay is related to how many of the triplets (CTM) of the considered design are kept in subsequent versions or releases. As a software evolves, additions of all sorts are to be expected, as new requirements and new functionalities will be implemented in the software system. In contrast, deletions are less "natural" and more likely to be associated with the correction of early misconceptions, and related to design decay. The absence of those relations in subsequent versions is an interesting measure of design decay. In particular, there may be cases in which the classes are kept but the relations between them are deeply modified.

The second measure considers the number of triplets in the tunnel (TTM). For this preliminary study, we are only interested in evaluating how much of an original design is present throughout a life time. To this end, we count the number of triplets in the tunnel (TTM) of each system, to measure their design decay. If TTM decreased, then the original design decayed. If TTM is stable, then the original design is stable, which means that the system is more adapted to the new changing requirements. For example, as illustrated in Figure 3.7 and 3.9, the tunnel of JFreechart decreased faster than the tunnel of XercesJ over the n versions, which means the structural changes are more frequent in JFreechart than in XercesJ. In both systems, numbers of triplets in the tunnel become stable.

RQ4: Do stable and decaying micro-designs have the same risk to be bug-prone?

Table 3.5 – 3.6 present 2×2 contengency tables for ArgoUML and Rhino. These tables report the number of (1) unstable classes, belonging to decaying micro-designs ($D\mu_D$), that are identified as bug-prone; (2) unstable classes that are identified as clean; (3) stable classes, belonging to stable micro-designs ($S\mu_D$), that are identified as bug-prone; and, (4) stable classes that are identified as clean. The result of Fisher's exact test and odds ratios when testing H_0 are significant. The p-value is less then 0.05 and the odds ratio for fault-prone unstable classes is two times higher than for fault-prone stable classes.

	Bug-prone classes	**Clean classes**
$\mathbf{D\mu_D}$	973	763
$\mathbf{S\mu_D}$	148	301
Fisher's test ($\mathbf{p-value}$)	$2.2e^{-16}$	
Odd-ratio (OR)	2.59	

Table 3.5 – Contingency table (ArgoUML) and Fisher test results for unstable classes with at least one bug.

We can answer **RQ4** as follows: we showed that stable micro-designs, belonging to the original design, are significantly less bug-prone than decaying micro-designs

	Bug-prone classes	**Clean classes**
$\mathbf{D\mu_D}$	105	14
$\mathbf{S\mu_D}$	39	17
Fisher's test (p – value)	0.005	
Odd-ratio (OR)	3.244	

Table 3.6 – Contingency table (Rhino) and Fisher's test for unstable classes with at least one bug.

and thus, we confirm previous findings [123].

RQ5: Are decaying micro-designs more prone to design defects than stable micro-designs?

Table 3.7 – 3.8 present 2×2 contengency tables for ArgoUML and Rhino. These tables report the number of (1) unstable classes, belonging to decaying micro-designs ($D\mu_D$), that are identified as prone to design defects; (2) unstable classes that are identified as clean; (3) stable classes, belonging to stable micro-designs ($S\mu_D$), that are identified as prone to design defects; and, (4) stable classes that are identified as clean. The result of Fisher's exact test and odds ratios when testing H_0 are significant. The p-value is less then 0.05 for ArgoUML and less than 0.06 for Rhino. The odds ratio for unstable classes that are prone to design design defects is three times higher in ArgoUML (respectively, two times in Rhino) than for stable classes that are prone to design defects.

	Design defect-prone classes	**Clean classes**
$\mathbf{D\mu_D}$	1305	431
$\mathbf{S\mu_D}$	210	239
Fisher's test (p – value)	$2.2e^{-16}$	
Odd-ratio (OR)	3.44	

Table 3.7 – Contingency table (ArgoUML) and Fisher's test for unstable classes with at least one design defect.

We can answer **RQ5** as follows: we showed that stable micro-designs, belonging to the original design, are significantly less prone to design defects than decaying micro-designs and thus we can confirm the results of the previous work [85].

	Design defect-prone classes	Clean classes
Dμ_D	95	24
Sμ_D	38	18
Fisher's test (p – value)	0.06327	
Odd-ratio (OR)	1.86	

Table 3.8 – Contingency table (Rhino) and Fisher's test for unstable classes with at least one design defect.

RQ6: How does ADvISE perform?

Table 3.9 shows the computation time in seconds of *ADvISE* for ArgoUML, DNSjava, JFreeChart, Rhino, and XercesJ. The average times for Step 2 (class renaming detection), Step 3 (bit-vector matching of two system versions), and Step 4 (clustering algorithm) were less than 3 seconds. The median time of all those steps were less than one second. Overall, the matching process (including PADL and EPI) took less than one minute for small sized systems, 2 minutes for medium-sized systems, and about 5 hours for a large-sized system.

Systems	Releases	Step 1 (PADL)	Step 1 (EPI)	Step 2	Step 3	Step 4	Step 5
ArgoUML	v0.10.1	7.047	18,098	4.835	10.651	10.140	908.329
DNSjava	v1.2.0	2.249	44.209	0.862	0.935	0.075	7.150
JFreeChart	v0.5.6	2.197	62.268	3.135	1.907	0.099	50.030
Rhino	v1.4.R3	2.150	50.350	1.783	0.450	0.064	7.985
XercesJ	v1.0.1	4.520	179.41	1.273	0.549	0.032	15.488
Median Time		2.249	62.268	1.783	0.935	0.075	15.488
Average Time		3.6326	3,686.84	2.377	2.898	2.082	197.7964

Table 3.9 – Execution time (in seconds) for each step of ADvISE.

Threats to Validity

Several threats potentially impact the validity of our study.

Construct validity threats concern the relation between theory and observation; in our context, they are mainly due to errors introduced in measurements. Our strategy of reverse engineering class diagrams may contain imprecision and there is a need to compare obtained results with other reverse engineering tools. Nevertheless, because all class diagrams were produced by the same tools chain, any imprecision should factor out. However, we can not exclude the possibility that by using a different reverse-engineering tool our algorithms may produce slightly different results. Another critical element is the faults data sets (bugs and design defects). We use manually-validated faults that have been used in previous studies [16][17]. Yet, we cannot claim that all fault-prone classes have been correctly tagged or that fault-prone classes have not been missed. There is a level of subjectivity in deciding if an issue reports a fault and in assigning this fault to classes. In this context, we are just interested to check whether a class is faulty or not, rather than quantifying the amount of faults (which is our future work).

Internal validity threats do not affect this particular study, being an exploratory study. Thus, we cannot claim causation, but just relate decayed classes with the occurrences of faults, although our discussion tries to explain why some decayed classes could have been subject to faults.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the statistical tests that we used (we mainly used Fisher test, which is a non-parametric test).

External Validity threats relates to the extent to which we can generalise our results. The main threat to the external validity of our study that could affect the generalisation of the presented results relates to the analysed systems. We performed our study on five different Java systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalised to other larger systems and systems in other programming languages. Future work includes replicating this study on other systems to confirm our results.

3.4 Summary

Design decay is defined as the deviation from an original design, *i.e.*, the violation of design caused by the process of evolution [63, 100, 127]. When evolution occurs in an uncontrolled manner, software systems become more complex over time and, thus, harder to maintain [12, 55]. Thus, decayed designs make systems more prone to bugs [123]. To the best of our knowledge, no previous approach exists to quantify and study design decay.

[16]`http://www.cs.columbia.edu/~eaddy/concerntagger/`

[17]`http://www.ptidej.net/downloads/experiments/emse10`

In this chapter, we propose a novel approach, $ADvISE$, and a set of measures (TTM and CTM) to measure design decay. The first step in observing design decay is to use a diagram matching technique to identify structural changes among versions of designs. Finding structural changes occurring in long-lived evolving designs requires the identification of class renamings. The second step requires to match evolving designs to identify stable/unstable triplets and thus, to identify stable/decaying micro-designs. The third step consists of using the previously-identified stable triplets in proposing metrics (TTM and CTM) to measure the design decay.

Thus, this chapter presented three contributions:

1. The first contribution of this chapter is a set of structural and textual similarities to identify class renamings in evolving designs.
2. The second contribution are a bit-vector and incremental clustering algorithms to perform the matching between several versions of a design and find stable/decaying micro-designs.
3. The third contribution is a set of metrics (TTM and CTM) that could be used as indicators of decay in the context of an evolving design, and thus, predictors of design defect- and bug- proneness.

We also performed a qualitative and a quantitative studies to show the applicability and usefulness of our approach. We applied our approach on five open-source systems: ArgoUML, DNSjava, JFreeChart, Rhino and XercesJ, and answered the following research questions as follows:

- **RQ1: What are the thresholds for class renaming detection?** We show that we can systematically choose adequate thresholds that provide an optimal F-measure (precision and recall) for our renaming detection technique.
- **RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?** We show that our approach has good precision and recall for class renamings detection.

- **RQ3: What are signs of design decay and how can they be tracked down?** We show that our design decay metrics (TTM and CTM) provide us useful insights regarding the signs of software aging. If TTM decreased, then the original design decayed. If TTM is stable, then the original design is stable, which means that the system is more adapted to the new changing requirements. If CTM increased, then new requirements and new functionalities are implemented in the software. If CTM is stable, then the system is stable and the most of maintenance activities are bug fixes.

- **RQ4: Do stable and decaying micro-designs have the same risk to be bug-prone?** We show that stable micro-designs, belonging to the original design, are significantly less bug-prone than decaying micro-designs.

- **RQ5: Are decaying micro-designs more prone to design defects than stable micro-designs?** We show that stable micro-designs, belonging to the original design, are significantly less prone to design defects than decaying micro-designs.

- **RQ6: How does ADvISE perform?** We show that the time performance of our approach is good, outlining the execution time of each step of $ADvISE$.

CHAPTER 4

A SEISMOLOGY-INSPIRED APPROACH TO STUDY THE CHANGE IMPACT

Although object-oriented programming has met great successes in modeling and implementing complex systems, developers face problems with maintenance [103]. In particular, making changes without understanding their effects can lead to poor effort estimation and delays in release schedules because of their dire consequences, *eg.*, the introduction of bugs [12, 55]. Therefore, both managers and programmers must be aware of the ripple effects caused by a change. Thus, they need help to identify the classes that must be changed to perform maintenance changes more accurately. Change impact analysis aims at identifying software artefacts being affected by a change; it provides the potential consequences of a change and estimates the set of artefacts that must be modified to accomplish a change [17].

The following *motivating example*, illustrates the difficulty that developers face in identifying the change impact: the bug ID200551[1] reports a bug in Rhino[2], that was introduced by a developer when he implemented a change to class `Kit` and missed a required change to class `DefiningClassLoader`. In this case, information passes from class `Kit` to class `DefiningClassLoader` through an intermediary class `ContextFactory`. Thus, a change in `Kit` should trigger a change in `DefiningClassLoader`, while the class `ContextFactory` remains unchanged.

This motivating example presents a situation where a bug was introduced by a developer who missed changing a class, that must be considered before implementing the change task. This example confirms that change impact is difficult to anticipate between two classes separated by an intermediary class. This problem would be more difficult if the two classes (`Kit` and `DefiningClassLoader`) were separated by a long chain of relations.

Studying the change impact, or more specifically the scope of change propagation, could help developers prioritise their changes according to the forecast

[1]`https://bugzilla.mozilla.org/show_bug.cgi?id=200551`

[2]`http://www.mozilla.org/rhino/`

scope of changes. Understanding change propagation requires source code analysis, which is a difficult, error-prone, and expensive activity [101]. We propose an approach to change propagation analysis specifically designed to study the scope of change propagation, based on a metaphor between seismology and change impact analysis. Our approach considers changes to a class as an earthquake that propagates through a long chain of intermediary classes. It combines static dependencies between classes and historical co-change relations to study the scope of change propagation in a system, *i.e.*, how far a change propagation will proceed from an "epicenter class" to other impacted classes.

In this chapter, we perform a qualitative and two quantitative studies, to show the applicability and usefulness of our approach. We apply our approach on three open-source systems: Pooka, Rhino, and XercesJ, and answer the following research questions:

- **RQ1: Does our metaphor allow us to observe the scope of change propagation?**
- **RQ2: What is the level most impacted by a change?**
- **RQ3: What is the most reachable level by a change?**

We answer these research questions as follows:

- **RQ1**: Like earthquakes the change impact seems to be more severe near the epicenter class and decreases through class levels.
- **RQ2**: We applied $ANOVA$ and $Duncan - Multiple - Range$ tests, and we can conclude that: a) level 1 is the most impacted, b) level 2 is the second most impacted, but significantly less than level 1, c) levels 3, 4, 5, and 6, are significantly less impacted than level 1 and 2.
- **RQ3**: We conclude that almost change propagation stops at the level 1 and 2. But, there are some earthquakes that propagate until 3, 4, 5 or 6 level.

This chapter is organised as follows: Section 4.1 describes our metaphor and mapping. Section 4.2 presents our approach and its implementation. Section 4.3

presents the three research questions derived from our metaphor while Section 4.4 presents our study results and answers to the questions. Finally, Section 4.7 concludes and presents future work.

4.1 The Earthquake Metaphor

We now present a mapping between the concepts of seismology and change impact analysis. We use this mapping to observe and identify the scope of change propagation, *i.e.*, how a change to a system will impact the rest of the system, using seismology techniques.

4.1.1 Seismology

Seismology is the study of earthquakes and of the propagation of seismic waves. A seismic wave, or shaking, is the vibration that occurs from the epicenter of an earthquake until a damaged site. Seismic waves propagate along the surface and through the Earth at varying speeds, depending on the types of soil through which they move. In general, shaking is most severe near the epicenter and decreases away from the epicenter, *i.e.*, more a site is near to the epicenter, more the debris are important [26]. Seismology includes three main research directions:

1. **Earthquake predictions**: Creating effective approaches for precise earthquake predictions, *i.e.*, forecasting the probable timings, locations, and magnitudes of earthquakes [67, 91].
2. **Debris forecasting**: Predicting and estimating debris, or structural damage, after an earthquake, to assist debris managers in planning large scale debris removals[3].
3. **Earthquake behaviour analysis**: Studying short- or medium-term interactions among earthquakes (fore shock, main shock, after shock), and long-term behaviour of earthquakes[109].

[3]`http://www.calema.ca.gov/WebPage/OESWebsite.nsf/Content/88892A0B623B1F77882574270081DD56?OpenDocument`

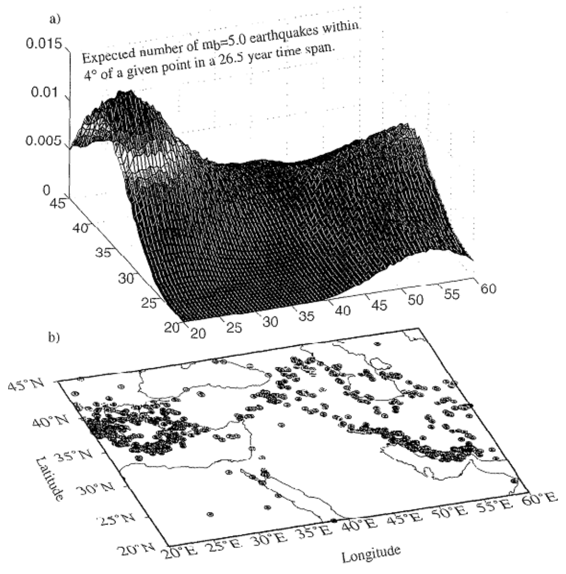


Figure 4.1 – Epicenters distribution in space and time. The map shows the expected number of earthquakes of a given magnitude occurring within a given radius from each point (From [95]).

Our approach is inspired from debris forecasting to identify where the impact is located (*i.e.*, the most risky classes). Figure 4.1 illustrates how the magnitude of an earthquake varies in space and time. We see that the magnitude is maximum in the epicenter and it decreases in function of the distance from the epicenter to the considered site.

4.1.2 Change Impact Analysis

Change impact analysis has two main goals: supporting the processing of changes and enabling the traceability of changes. It is important during development and maintenance to help developers in assessing their effort to implement change requests (typically, the more impacted classes by a change, the greater the effort) and in performing the most adequate changes. Thus, it limits the risk of introducing bugs by clearly identifying classes that could be impacted.

Change propagation begins with a class being changed. This change propagates and forces other classes to change. These changed classes, in turn, may yield to

Change Impact Analysis	**Seismology**
“Important” classes	Active seismic areas
Software change	Earthquake
“Important” changed class	Epicenter
Change propagation	Seismic wave propagation
“Other” changed classes	Damaged sites
Class level	Distance from an epicenter to a damaged site

Table 4.1 – Mapping between Change Impact Analysis and Seismology.

other changes. Thus, change propagation is due to changes “moving” from one class to another through system classes. For example, if a method is renamed because of some change request, then all the classes that call this method must be modified to use its new name; in turn, other classes may change because of the changes in these methods.

4.1.3 Change Impact Analysis and Seismology

We now define a mapping between impact analysis and seismology, summarised in Table 4.1.

In seismology, the epicenter of an earthquake is located in a most active seismic areas. To determine the epicenter (location) of an earthquake, seismologists analyse seismograms, which record the seismic activities for a given areas.

In change impact analysis, any class is potentially subject to changes. Yet, changes to “important” classes will impact more a system than “peripheral” classes. A class can be characterised as important according to different measures. If an “important” class changes, then it is analogous to an epicenter in seismology. In the following Section 4.2, we identify important classes and also apply our approach to all the classes in three systems.

In seismology, seismic waves propagate from the epicenter of an earthquake until a damaged site, depending on the types of soil through which they move. Shaking is usually most severe near the epicenter and drops off away from the epicenter, *i.e.*, more the infrastructures are near the epicenter more the debris is important.

In change impact analysis, changes propagate from the epicenter class to the impacted class, depending upon the class relations, other logical couplings among

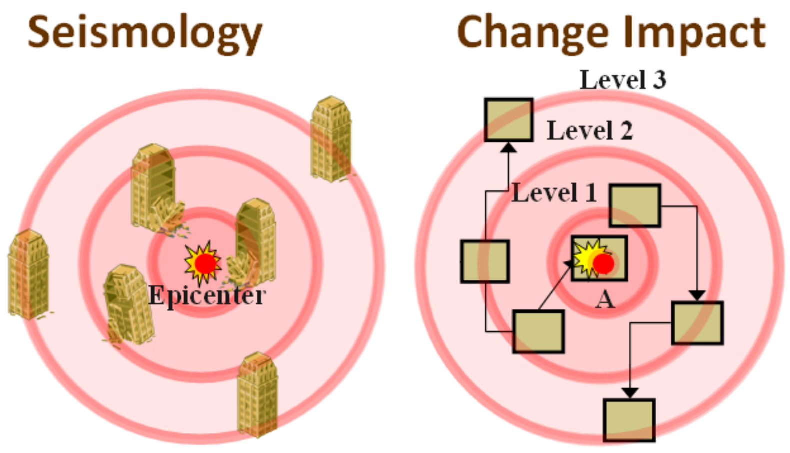


Figure 4.2 – Earthquake Metaphor.

these classes, and their distance (called class level) to the epicenter class. We define the distance between classes using the concept of class level: with respect to a class A, a class B is in the level 1, if it is in direct relation with A (inheritance, call, etc), and in the level 2, if it is related to A through an intermediary class C.

Change propagation is analogous to seismic waves propagation in seismology. As in seismology, we assume that change impact is most severe near the epicenter class, *i.e.*, more the classes are near to the epicenter class more the impact is important (see Figure 4.2). Yet, to the best of our knowledge, no previous work reported observations on the propagation of changes through the systems, which is the aim of our metaphor and the subject of our empirical study. Using this mapping between impact analysis and seismology, we now present an approach to identify the classes impacted by a change to a given class. This approach is complementary to previous work. It uses structural and historical analysis with the specific aim to study the scope of change propagation.

4.2 Approach

This section presents our approach, each step will be described in detail below. We can apply our approach on any class. Specifically, as a developer starts changing a class, it could analyse the change and recommend additional classes for consideration. Also, before performing any change in a system, it could help a developer to identify critical classes that are regularly changed and could have an impact on the system.

Our approach consists of three steps. Given an object-oriented system, first, we compute metric values to rank classes according to their importance and identify "epicenter classes ". Second, given an epicenter class, we compute the "distance" from the epicenter class to each class of the system, using a bit-vector algorithm: all classes that are directly connected to the epicenter class are assigned to the level 1, the classes that have a direct relation with one of these classes are put in level 2, and so on. Third, we use a time window of duration T and collect the set of classes that are modified after any change of the epicenter class and within T. Finally, we report the set of classes that are involved in any change and their number of changes.

4.2.1 Step 1: Measuring Class Importance

In this Step, we identify the most "important" classes in a system using a combination of history-based and PageRank-based metrics.

A. History-based Metric

We define a history-based metric as the number of all commits related to a given class in the entire history of a system, extracted from the system version-control system. This measure represents the quantity of changes to a class. It does not consider the size and the type of a change, which are future work.

We use Ibdoos, our framework for the analysis of control-version systems, to compute the numbers of changes to every classes in a system. Ibdoos provides parsers for various format of change logs, including CVS and SVN, and stores all commits in a database, then we query this database to obtain the numbers of changes per classes. We define the number of changes to a class as $h(c)$ for any class c.

B. PageRank-based Metric

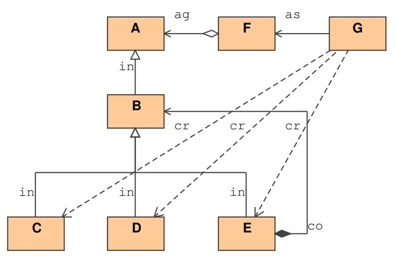
PageRank [98] is one of the main algorithms used by the Google search engine to measure the relative importance of Web pages. PageRank takes backlinks into account and propagates the PageRank value of a page through links: a page becomes important if the sum of the values of its backlinks is high. Using PageRank, we can measure the relative importance of each class in a system. A class is important if it has incoming calls from other (preferably important) classes. If a class is the only reference of a very important class, it might be ranked higher than another class in relation with low-ranked classes. We use the algorithm previously developed by Kpodjedo [75] to compute for each class c, the PageRank value $pr(c)$. Then, we rank all classes in descending order of their $pr(c)$ and we compute their class rank value $r(c)$: the most important class c has the highest $pr(c)$ value.

C. Combination of the History- and PageRank-based Metrics

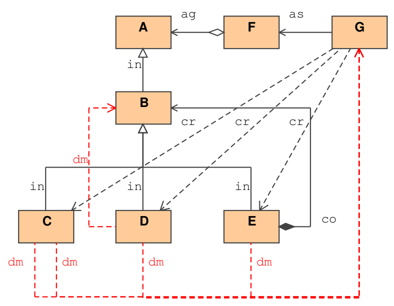
We combine the history- and PageRank-based metrics by dividing the rank of each class by the number of changes to the class. Thus, given two classes with equal ranks, the most important class of the two is the one with the greater number of changes, which lead to its rank to become higher than that of the other class. We thus define $rh(c) = \frac{r(c)}{h(c)}$.

The metric rh provides an assessment of the class importance in a system, taking into account both the system structure (through the PageRank-based metric) and the system history (through the history-based metric). The combination rh ranks the most important classes first. A lower value of $rh(c)$ indicates a lower value $r(c)$ and a higher value $h(c)$, *i.e.*, rh ranks the most important classes that are often changed. Identifying these important classes helps reveal what classes of the system are regularly evolved and should be analysed to identify their change propagation.

(a) UML-like model



(b) Eulerian model



(c) String representation of the Eulerian model

```
A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A
```

Figure 4.3 – Representations of a simple example system (from [66])..

4.2.2 Step 2: Identifying Class Levels

We assume that change propagation depends on the “distance” between classes (called level). We represent a level as the number of the intermediary relations between a given class and the epicenter class, *i.e.*, this distance indicates whether the two classes are in direct relation (level 1) or are separated by a long chain of relations of length n.

Again, we use an existing tool, PADL [52], to automatically reverse-engineer class diagram from the source code of object-oriented systems. We recall that a model of a system is a graph with nodes being the classes and edges representing the relations between classes, see Figure 4.2(a). To identify direct and indirect relations with the epicenter class, we first convert the system model into its string representation, as illustrated in Figure 4.2(c), using the algorithm previously developed by Kaczor [66]. Then, we apply a bit-vector algorithm [13]: we build the characteristic vectors of each token in the string representation. The characteristic vector of a token t associated with the string $s = s_1...s_m$, is $t = (t_1...t_m)$:

$$t_i = \begin{cases} 1 & \text{if } s_i = t \\ 0 & \text{otherwise.} \end{cases}$$

For the example shown in Figure 4.3, the characteristic vectors of tokens A, in, and B are defined as:

$$\begin{aligned} A &= \mathbf{1}\underbrace{0000000000000000}_{30}\mathbf{1} \\ in &= 0\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{0000000}_{19} \\ B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000}_{20} \end{aligned}$$

Lets assume a class A as an epicenter class, to identify all classes that are directly connected to this epicenter class: for each class X—using conjunctions and shifts—between the characteristic vectors of A, X, and all relations: if X is directly related to A through a relation, then we put X in level 1. For example, to identify whether class B is directly related to class A through the inheritance in, we compute:

$$\begin{aligned} (\rightarrow\rightarrow A) &= 0\mathbf{11}\underbrace{0000000000000000000}_{29} \\ (\rightarrow in) &= 00\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{000000}_{18} \\ B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000}_{20} \\ R &= (\rightarrow\rightarrow \mathbf{A}) \wedge (\rightarrow \mathbf{in}) \wedge B \\ &= 00\mathbf{1}\underbrace{00000000000000000000}_{29} \end{aligned}$$

and assess whether the bit vector R is null (contains only zeros). If R is not null, then class B is in level 1 with respect to A. Once all classes of level 1 are defined, we repeat this process, considering each class of level 1 as epicenter class to identify all classes at level 2, *i.e.*, classes that have a direct relation with a class of level 1. The same process is repeated to identify level 3, and so on.

4.2.3 Step 3: Identifying Impacted Classes

We mine the version-control system of a system to identify epicenter classes: we first define a time window T of observation as the median of time between two subsequent changes to the important epicenter class. We choose the median because it is robust to outliers. Then, we extract all the commits that happened after any change to an epicenter class and within the chosen time window T. Finally, we collect: (1) the names of all classes that are involved in any change and (2) the number of changes to each class.

We use our framework Ibdoos to implement queries to collect the set of classes changed after any change to the epicenter class and during T. The names of all subsequently changed classes and the number of changes that these classes underwent.

4.3 Empirical Study Design

Following the Goal Question Metric (GQM) [10], the *goal* of this study is to show the applicability and usefulness of our approach, with the *purpose* of gathering interesting observations on the scope of change propagation and confirming these observations statistically. The *quality focus* is the accuracy of the identified scope of change propagation (*i.e.*, how far the propagation proceed from a given class to the others), and also the variation of changes depending on the level of classes. The *perspective* is of both researchers and practitioners who should be aware of the scope of a change to estimate the effort required for future maintenance tasks. The observed phenomena can help for making decisions concerning the process of future software projects. The *context* consists of Pooka, Rhino, and XercesJ. Pooka[4] is an email client written in Java, using the Javamail API. Rhino[5] is an open-source implementation of a JavaScript interpreter written entirely in Java

[4]`http://www.suberic.net/pooka/`
[5]`http://www.mozilla.org/rhino/`

Systems	Nbr. Classes	Start Dates	End Dates	Last Version
`Pooka`	298	2000-01-02	2010-08-30	2.0
`Rhino`	132	1999-12-18	2009-01-17	$1R6.0$
`Xerces`	685	2005-10-12	2010-11-26	11.0

Table 4.2 – Statistics for the systems..

and developed for the Mozilla/Firefox browser. Xerces[6] is an open-source family of software packages for parsing and manipulating XML. Characteristics of these systems are reported in Table 4.2. We choose these systems because they have been studied in previous work, they are open-sources, thus we can find external information, such as bug reports.

4.3.1 Research Questions

This study aims at answering the research questions:

- **RQ1: Does our metaphor allow us to observe the scope of change propagation?**

  We investigate whether it is possible to apply our approach to observe change propagation through class levels. We perform a qualitative study to confirm our observations of change propagation, using external information. Thus, we can show that, indeed, like in seismology, certain levels are more impacted by a change than others.

- **RQ2: What is the level most impacted by a change?**

  We perform a quantitative study to confirm our observations of change propagation, using statistical tests to investigate which level may be the most impacted by a change, and classifying the levels having similar impact. Thus, we can deduce all classes with a higher risk to be impacted by any change to epicenter class.

- **RQ3: What is the most reachable level by a change?**

  As in **RQ2**, we perform a quantitative study to confirm our observations of change propagation, using statistical tests to investigate, for each level,

[6]`http://xerces.apache.org/`

the number of earthquakes that propagate until a given level. Thus, we can deduce the most reachable level.

4.3.2 Analysis Methods

To answer RQ1, RQ2, and RQ3, we apply our approach on Pooka, Rhino, and XercesJ.

RQ1: Does our metaphor allow us to observe the scope of change propagation?

As in seismology, we are interested in the most important seismic sources. Thus, we use metrics combination $rh(c)$ to rank classes according to their importance. We observed the two most important epicenter classes in Rhino, and XercesJ. To illustrate our observations, we selected four representative epicenter classs, in Rhino, and XercesJ, as shown in Table 4.3 and Table 4.4. For each epicenter class, we report information from the systems bug trackers and mailing lists confirming the propagation of a change to other classes in different levels.

Using the R statistical system[7], we build the 3D graph visualising the change propagation from the epicenter class to other classes, through their levels. The axes of the graphic are the time, levels, and numbers of changes. Thus, we study the graph of a representative epicenter class in each system to assess whether, as in seismology, change impact is most severe near the epicenter and decrease far away from the epicenter.

RQ2: What is the level most impacted by a change?

We perform an exhaustive study to confirm our observations, using a statistical test, for all classes in Pooka, Rhino, and XercesJ. We consider each class as an epicenter class. We compute, for each level, the number of classes that changed after any change to the considered epicenter class and within the chosen time window.

For each level, we create a subset that contains the number of changes per class. We then apply ANOVA on these subsets to determine whether there are significant

[7]`http://www.r-project.org`

Epicenter Classes	$pr(c)$	$r(c)$	$h(c)$	$rh(c)$
`Context`	0.043243	2	135	67.5
`IdScriptableObject`	0.057838	1	9	9
`Kit`	0.038378	3	14	4.66
`BaseFunction`	0.027838	7	37	5.28

Table 4.3 – Epicenter classes in Rhino.

Epicenter Classes	$pr(c)$	$r(c)$	$h(c)$	$rh(c)$
`TypeValidator`	0.020261	3	25	0.12
`XMLEventImpl`	0.017062	6	21	0.28
`DeferredDocumentTypeImpl`	0.006441	25	68	0.36
`XMLEntityScanner`	0.002602	76	194	0.39

Table 4.4 – Epicenter classes in XercesJ.

difference between subset means. When differences between subsets exist, the null hypothesis "H_0: the number of changes is similar for each level" is rejected. Then, we conduct Duncan's multiple range test to classify the subsets with respect to the differences between them. We choose Duncan's multiple range test because it can maintain a low overall type I error [14] and it uses a studentized range statistics within a multiple range test. We interpret the range-value as follow:

- Range 1, if subsets mean value is the minimum.
- Range 2, if subsets mean is adjacent to Range 1 mean.
- Range 3, if subsets mean is adjacent to Range 2 mean.

RQ3: What is the most reachable level by a change?

This research question aims to verify whether most earthquakes would propagate through all class levels or just the first level. For each level, we create a subset that contains the number of earthquakes that stop at this level. We then apply again ANOVA on these subsets to determine whether there are significant differences between their means. When differences between subsets exist, the null hypothesis "H_0: the number of earthquakes is similar for each level" is rejected. If the ANOVA results yield significant differences, we again apply Duncan's multiple range test to classify the subsets with respect to their differences. For example,

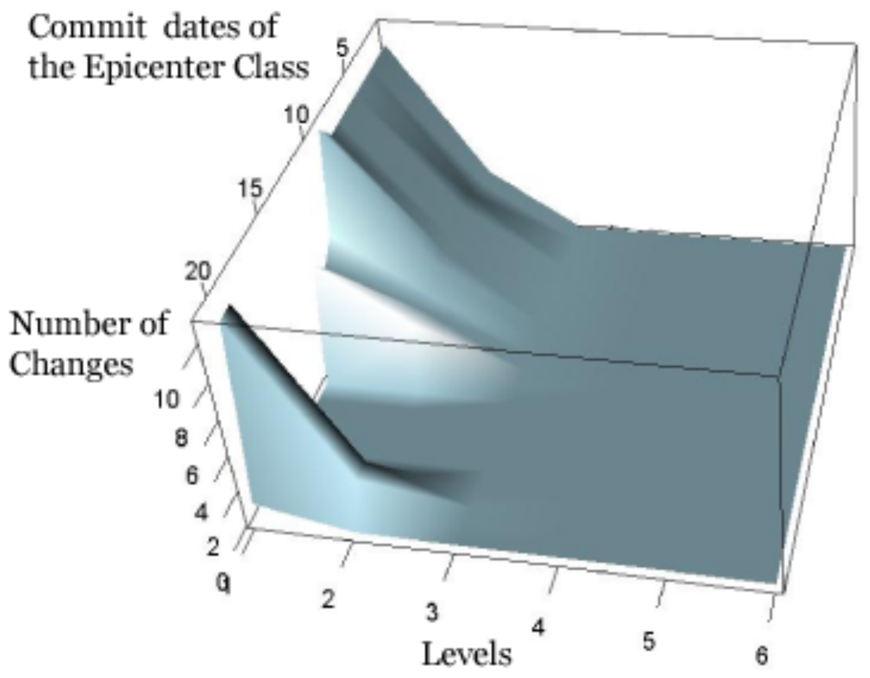


Figure 4.4 – Change propagation from XMLEventImpl.

in XercesJ, the most reachable level by the epicenter class `XMLEventImpl` is level 3, while the epicenter class `TypeValidator` reaches at maximum level 6.

4.4 Empirical Study Results

We now present the results of our empirical study.

RQ1: Does our metaphor allow us to observe the scope of change propagation?

We answer positively to this question using the epicenter classes selected in Rhino, and XercesJ.

In one hand, we build the 3D graphs visualising the change propagation from the selected epicenter classes in Pooka, Rhino, XercesJ. Due to the lack of space in this chapter, we present just two graphs in XercesJ. Figure 4.5 and Figure 4.4 illustrate the change propagation phenomena through levels for a specific epicenter

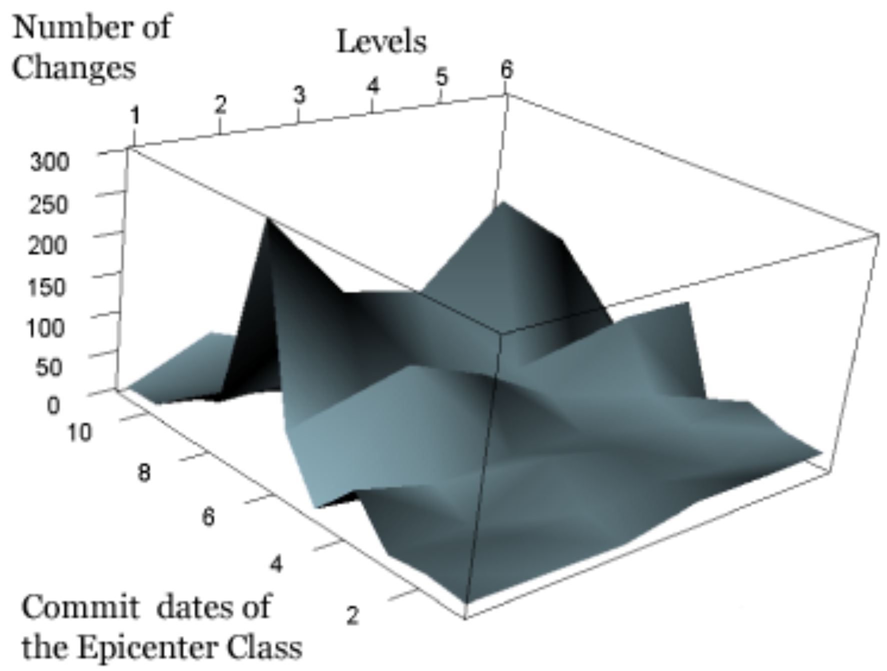


Figure 4.5 – Change propagation from TypeValidator.

class commit, in XercesJ, for the epicenter classes `XMLEventImpl` and `TypeValidator`. In `XMLEventImpl`, number of changes is very important for the first level, after that it decreases through the $2nd$ and $3rd$ levels. In `TypeValidator`, we observe that changes propagate until the $4th$ level and then it decreases significantly. However, in some cases, they may propagate to $6th$ level. Thus, we conclude that change propagation is different

In the other hand, we found external information in the system bug trackers and mailing lists. Thus, we report four examples (in Rhino and XercesJ) of external information illustrating the propagation of a change from the chosen epicenter class to another class (in different level).

In Rhino

- Epicenter class `IdScriptableObject`: we found the bug ID256321[8] that confirms that a change to the epicenter class propagated to `ScriptableObject` (level 1) to make Rhino objects serialisable.

- Epicenter class `BaseFunction`: we found the bug ID236117[9] that reports that a change to the epicenter class propagated to classes `Context` and `ScriptRuntime` (level 1), and `ContextFactory` and `WrapFactory` (level2).

- Epicenter class `Context`: we found the bug ID255891[10] that relates the epicenter class to three classes of level 1 (`CompilerEnvirons`, `ContextFactory`, and `ScriptRuntime`).

- Epicenter class `Kit`: we found the bug ID200551[11] that reports that a change to the epicenter class propagates to the class `DefiningClassLoader` (level 2).

In XercesJ

- Epicenter class `TypeValidator`: we found a message[12] in the mailing list stating that changes to `TypeValidator` propagate to `PrecisionDecimalDV`.

- Epicenter class `XMLEventImpl`: we found a SVN log commented in mailing list[13][14][15] that confirms the change propagation from the epicenter class to the classes: `EndDocumentImpl`, `EntityReferenceImpl` (level 1), and `StartElementImpl` (level 2).

- Epicenter class `DeferredDocumentTypeImpl`: we found the on-line discussion[16] showing that changes to the epicenter class must propagate to four classes in

[8]https://bugzilla.mozilla.org/show_bug.cgi?id=256321
[9]https://bugzilla.mozilla.org/show_bug.cgi?id=236117
[10]https://bugzilla.mozilla.org/show_bug.cgi?id=255891
[11]https://bugzilla.mozilla.org/show_bug.cgi?id=200551
[12]http://comments.gmane.org/gmane.text.xml.Xerces.devel/5855
[13]http://xerces.markmail.org/message/eskpra4vcmaugtx6?q=XMLEventImpl+StartElementImpl
[14]http://xerces.markmail.org/message/33uxkvhfomocrngj?q=XMLEventImpl+StartElementImpl
[15]http://xerces.markmail.org/message/3hkhyuwj6v5oa7hw?q=XMLEventImpl+StartElementImpl+&page=2
[16]http://xerces.markmail.org/search/?q=DeferredDocumentTypeImpl#query:DeferredDocumentTypeImpl+page:2+mid:iyb37kwel5rdmaod+state:results

	Homogenous subsets for alpha = 0.1		
Levels	Range 1	Range 2	Range 3
5	107.5410		
4	147.7778		
3	150.0000		
2		202.0408	
1			354.4828

Table 4.5 – Duncan's test applied on "number of changes of the impacted classes" in Rhino.

	Homogenous subsets for alpha = 0.1		
Levels	Range 1	Range 2	Range 3
6	6.4015		
5	10.8485		
4	24.8333		
3		50.2789	
2		83.7273	
1			895.2652

Table 4.6 – Duncan's test applied on "number of changes of the impacted classes" in XercesJ.

level 2: `DeferredElementImpl`, `DeferredEntityImpl`, `DeferredNotationImpl`, and `DeferredTextImpl`.

- Epicenter class `XMLEntityScanner`: we found the bug ID1099[17] that relate the changes to the epicenter class with changes to `XMLParser` (level 3).

RQ2: What is the level most impacted by a change?

We apply our approach on Pooka, Rhino, and XercesJ. Then, we create subsets that contains numbers of changes per class that propagates from each class, then we apply ANOVA on these subsets. The ANOVA results yield significant difference between subset means. We therefore conduct Duncan's multiple range to classify these subsets in each three systems. Table 4.5 summarises the results of Duncan's test applied on Rhino. We observe that change propagation, in some cases, reach level 5. This table shows that all the sample means are significantly different for levels 1 and 2 (because they are classified in different ranges), except the means of levels 3 and 4, for which there is no evidence of a difference, and thus they are grouped together in the same range. The non significant difference between levels

[17]`https://issues.apache.org/jira/browse/XERCESJ-1099`

3 and 4 suggests that the number of changes are similar. The number of changes is much higher in level 1 (corresponding to the mean value 354.4828, and range 3) and this high difference (with respect to other means) results in a separate range. The same goes for the second level (corresponding to the mean value 202.0408, and range 2).

Table 4.6 summarises the results of Duncan's test applied on XercesJ. We observed that change propagation, in some cases, reach level 6. This table shows that level 1 differs significantly from the others, by being the most impacted. The levels 4, 5 and 6 are classified in range 1, thus the number of changes is almost similar at these levels. But, they are less impacted than levels 2 and 3 that are classified in range 2 (corresponding to the means values 50.2789 and 83.7273). The number of changes is much higher in level 1 (corresponding to the mean value 895.2652).

From the results of the three systems, we can conclude that: a) level 1 is the most impacted, b) level 2 is the second most impacted, but significantly less than level 1, c) levels 3, 4, 5, ... are significantly less impacted than level 1 and 2, and d) levels 3, 4, 5, ... are classified in the same range, because the number of changes seems similar in these levels.

RQ3: What is the most reachable level by a change?

	Homogenous subsets for alpha = 0.1		
Max Level	Range 1	Range 2	Range 3
5	.5833		
4	1.3712		
3	1.7500		
2		4.6136	
1			11.7121

Table 4.7 – Duncan's test applied on "number of earthquakes" in Rhino.

In this research question, we apply the same approach as for RQ2. Here, for each level, the subset is the number of earthquakes that stop at this level. The ANOVA results yield significant differences. Thus, we apply Duncan's test.

Table 4.7 summarises the results of Duncan's test applied on Rhino. We observe that the number of earthquakes that reach at maximum level 1 is greater (corresponding to the highest mean 11.7121) than those that reach the other levels. But,

	Homogenous subsets for alpha = 0.1		
Max Level	Range 1	Range 2	Range 3
6	10.5333		
5	16.3333		
4	21.6667		
3		30.0033	
2		43.2000	
1			54.8667

Table 4.8 – Duncan's test applied on "number of earthquakes" in XercesJ.

some changes propagate through levels 3, 4, and 5. The means of the number of earthquakes that reach levels 3, 4, 5 are too similar (corresponding means are: .5833, 1.3712, and 1.7500), consequently, they are grouped in the same range 1. The earthquakes that reach level 2 are less than those that reach level 1, but significantly greater than the others (levels 3, 4 and 5). As a result, earthquakes that reach levels 1 and 2 cannot be classified with earthquakes that reach other levels.

Table4.8 summarises the results of Duncan's test applied on XercesJ. We observed that the number of earthquakes that reach at maximum level 1 are the most frequent (corresponding to the mean 54.8667). But, the number of earthquakes that reach at maximum the levels 4, 5, and 6, are almost similar, and the least (corresponding means are 21.6667, 16.3333, and 10.5333). The number of earthquakes that reach at maximum levels 2 and 3, are significantly similar, and thus are regrouped in the same range.

Thus, we conclude that almost change propagation stops at the level 1 and 2. But, there are some earthquakes that propagate until 3, 4, 5 or 6 level.

4.5 Discussions

We now discuss our approach and its empirical study. With our approach, we analysed change propagation in three different systems belonging to different domains and with different sizes, and histories. We observed that changes did not propagate through different class levels with the same proportion in each system. We observe that the numbers of earthquake propagations that stops at the level 1 and 2 is the greatest. However, the number of earthquakes that stop at higher levels (3, 4, and 5) are the least. By determining what levels might have been affected by certain changes, we can help maintainers to rapidly pinpoint the source of a bug. Consequently, maintainer needs to only examine the indicated levels in priority instead of inspecting all the source code: our method is time saving.

We observed that some classes changes frequently and are changed periodically by developers. This observation could help developers be aware of classes that they should consider changing even if their changes is not directly linked to these classes.

4.6 Threats to Validity

Several threats potentially impact the validity of our study.

Construct validity concerns the relation between theory and observations. In this study, they could be due to the chosen time windows which may affect our observations. A too long time window would be misleading while a very narrow time window would not allow to observe interesting facts. Conservatively, we chose a class-dependent time window: the median of time between two subsequent changes, because the median is robust to extreme outliers and thus minimises spurious changes induced by too large time windows on classes connected to epicenter classes. However, we may not have used the most revealing time windows. More investigation is needed to better understand the role of time windows. Finally, it is possible, despite the confirmation using external sources of information, that some classes reported as impacted by a change did actually change for reason independent of the changes to the epicenter class classes.

Internal validity is the extent to which a treatment impacts the dependent variable. The internal validity of our study is not threatened because we have not manipulated the independent variable, extent of the change propagation.

External validity relates to the extent to which we can generalise its results. The main threat to the external validity of our study that could affect the generalisation of the presented results relates to the analysed systems. We performed our study on four different Java systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalised to other larger systems and other programming languages. Future work includes replicating this study on other systems to confirm our results.

Conclusion validity deals with the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. We applied ANOVA and Duncan's multiple range tests. ANOVA assumes that the data are normally distributed. We may have a problem of assuring this assumption. Thus, we improved our conclusion validity by increasing the risk of making a Type I error (increase the chance that we will find a relation when in fact there is not), we can do that statistically by raising the alpha level. For instance, instead of using 0.05 significance level, we use 0.1 as our cutoff point.

4.7 Summary

Change propagation analysis in object-oriented systems is important to estimate the effort required for future maintenance tasks. The observed phenomena can help for making decisions concerning the process of future software projects, and reducing the overall cost of source code inspection [101].

Existing approaches for change impact analysis are based on the software structure and use static, dynamic, textual, and–or historical analyses [7, 17, 21, 79, 80, 135, 139]. However, to the best of our knowledge, none of these approaches have been used to study the scope of change propagation.

In this chapter, we proposed an approach to analyse change propagation and to study how far a change propagation will proceed from a given class to the others. Our approach considers changes to a class as an earthquake that propagates through the class levels, defined by the length of relations chain that relate the epicenter class to the other classes.

We performed a qualitative and two quantitative studies on three open sources: Pooka, Rhino and XercesJ, and thus, we answered the following research questions:

- **RQ1: Does our metaphor allow us to observe the scope of change propagation?**
- **RQ2: What is the level most impacted by a change?**
- **RQ3: What is the most reachable level by a change?**

We showed that our intuition, about the impacted classes by a change must be near to the changed class, is incorrect in some cases. Therefore, Duncan's multiple range test confirms that level 1 has the highest number of changes. However, there are some change propagations that reach the 5th level in Rhino (and 6th in XercesJ). Identifying the scope of change propagation could help, both developers and managers. Developers could locate easily the change impact, and thus they do not have to analyse the whole source code to understand the ripple effect of a change. They could include in their change set the classes belonging to the identified levels. Managers could estimate the efforts required to perform changes more accurately.

CHAPTER 5

AN IMMUNE-INSPIRED APPROACH FOR THE DETECTION OF DESIGN DEFECTS

Code smells [40] and antipatterns [23], collectively called in the following design defects, are bad solutions to recurring software design and implementation problems. They are conjectured to have a negative impact on the quality and life-time of systems [23, 40]. Consequently, their detection has received attention from both researchers and practitioners with approaches ranging from manual inspections to rule-based detection algorithms.

In this chapter, we present an approach to systematically detect classes whose characteristics violate some established design rules; rules inferred from sets of instances (*i.e.*, manually-validated occurrences) of defects reported in the literature and freely-available [70, 90]. Our approach detects design defects in general: although we train our approach on only three kinds of design defects, it can detect any number and any kind of design defects specified during the training. Moreover, it reports classes similar but not identical to the defects, which are of interest to developers and quality-assurance personnel.

Our approach stems from a parallel between object-oriented software systems and living bodies, which constantly fight invading pathogens, such as viruses, bacteria, and so on, through their immune system defense mechanisms. A natural immune system is able to protect the body by identifying, learning from, and defending against invading pathogens. It recognises pathogens after having fought the disease once or by the use of vaccines. Vaccines work by stimulating the immune system using small amounts of disactivated, disease-causing organisms. They cause the immune system to produce antibodies matching the pathogens. Antibodies react concretely to the presence of antigens carried by pathogens. Once antibodies are developed, the immune system is able to respond quickly to the infection of a similar or identical disease-causing organism entering the body, *i.e.*, pathogens carrying similar or identical antigens.

A useful parallel can be drawn from the natural immune system: a software design is comparable to a body, we wish to protect it from pathogens, such as design defects. Design defects detection approaches are defense mechanisms of the software design. A design defect is a pathogen. A "vaccine" could be build using instances of some defects, from which the software design should be protected. Occurrences of a defect are classes with characteristics similar or identical to the defect, *i.e.*, cells contaminated by some pathogen. Antigen that should trigger a response of the defense mechanism can be any characteristics of classes, *eg.*, metrics, binary class relations, and so on.

Like pathogens, defects come in a variety of forms with some defects being only slightly different from others. A natural immune system can handle such similar pathogens with good precision. This good precision is essential for the body and have inspired a family of classification algorithms name Artificial Immune Systems (AIS) algorithms. Oda and White commented that "if the immune system were inaccurate, the lifespan of the average human would be much shorter as the system would mistakenly attack vital cells or fail to attack viruses and other dangerous pathogens" [96]. Therefore, an AIS-based approach could potentially overcome the limitations of previous approaches regarding the detection of similar but not identical defects as well as the performance in time, precision, and recall of current state-of-the-art approaches. We propose a novel detection approach, called IDS (Immune-based Detection Strategy), based on AIS.

The chapter is organised as follows. Section 5.1 describes our approach. Section 5.2 presents the design of the experiments carried out to evaluate our approach. Section 5.3 reports and discusses the results. Section 5.5 concludes and suggests future work.

5.1 Artificial Immune Systems

5.1.1 Biological Background

Innate immunity defends the body from any pathogens that enter the body. Adaptive immunity allows the immune system to attack any foreign pathogens that the innate system cannot destroy. It can distinguish between the body's own cells and foreign cells.

The adaptive immune system is directed against specific invaders and is modified by exposure to such invaders. It is made up of *lymphocytes* (B cells and T cells). These cells aid in the process of recognizing and destroying specific antigens. Immune responses are normally directed against the antigen that provoked them and are said to be antigen-specific. The immune system learns to react to particular patterns of antigens. On the surface, each lymphocyte cell have receptors to recognize antigens, which are specialized: each can match only one specific antigen.

5.1.2 Computer Models

An artificial immune system (AIS) is a classification algorithm that mimics the immune system defense mechanisms. It can accept patterns of arbitrary length and it has the ability to maintain and exploit previously learned data efficiently for improved performance in future encounters with pathogens.

An AIS produces a large number of randomly-created *detectors* to recognise the antigens located on the surface of the foreign pathogens. A *negative selection* mechanism is applied to eliminate detectors that match the body's own cells. Kept detectors become *naive* detectors; they die after some time, unless they match some antigens; in case of such a matching, they become memory cells. Detectors that match a pathogen are quickly multiplied via the *clonal selection* to accelerate the response to further attacks. Also, because the clones are not exact copies (they are mutated, the mutation rate being an increasing function of affinity between detectors and antigens), they can both better focus on pathogens and handle similar pathogens (*affinity maturation*).

We draw a parallel between the immune system and the detection of defects. A software design is similar to a living body. It is protected from design defects by a detection approach, as a body is by its immune system. The detection approach identify defect classes, *i.e.*, pathogens, using some characteristics of the classes—in the following, metrics values—by comparing them to sets of metrics values of defect classes. Our novel detection approach, IDS (Immune-based Detection Strategy), can classify cells (system classes) that are present in the body (software design) as

Concepts of Immune System	
In Biology	In Software
Body	Software design
Immune system	Design defects detection approach
Self-Cells	Well-designed classes
Non-Self Cells (Pathogens)	Defect classes
Antigen	Sequence of metrics values
Antibody	Known pattern of metrics values
Affinity	Similarity measure between sets of metrics values
Negative Selection	Antibodies created during the learning phase

Table 5.1 – Instantiation of an AIS to detect design defects..

body's own cells (well-design classes) and foreign cells (defect-prone classes). We choose to characterise the body's and foreign cells with a set of metrics[1] [53].

5.1.3 Implementations

In general, any AIS algorithm has five steps: initialization, antigen training, competition for limited resources, memory cell selection, and classification [24]. The first step and the last step are applied only once, but Steps 2, 3, 4 are used for each sample (antigens) in the dataset.

Carter [29] developed the first AIS-based classification algorithm. It is a supervised learning system based on a high-level abstraction of *T cells*, *B cells*, *antibodies*, and an *amino-acid library*. The artificial T cells control the production of B-cell populations, which compete for recognition of the *unknowns*. The amino-acid library acts as a library of *epitopes* or characteristics of antigens currently in the system. When a new *antigen* is introduced into the system, its variables are entered into this library. The T cells then use the library to create their receptors that are used to identify the new antigen [119]. Brownlee [24] developed another algorithm, called Immunos-99, to combine the benefits of AIS-based classification algorithm with clonal selection classification algorithm [25]. Each step of this algorithm is explained below:

1: Divided data into antigen-groups (by classification labels)

[1]`http://www.ptidej.net/downloads/experiments/quatic10/`

```
2: Prepare a B-cell population
3: for each antigen-group do
4:     Create an initial B-cell population for an antigen-group
5:     for each generation do
6:         Create an initial B-cell population for an antigen-group.
7:         Calculate fitness scorings
8:         Perform population pruning
9:         Perform Affinity maturation
10:        Insert randomly selected Antigens of the same group
11:    end for
12: end for
13: Perform final pruning for each B-cell population
14: Present the final B-cell populations as the classifier
```

First, the algorithm divides the provided antigens into groups. Once prepared, a new B-cell population is created from each antigen group. The initial size of each B-cell population equals the number of antigens in the original group. The population is then exposed to all antigens, one antigen at a time and an affinity value is calculated for each B-cell to the antigen. The B-cell populations are then sorted in descending order of affinities. Once the training steps completed, the resulting B-cell populations form the classifier for new (invading) antigens. Each B-cell population is exposed to a new antigen and populations compete for "ownership" of the new antigen using their affinity. The population that have the highest affinity classifies the new antigen with its label.

Feature selection can be useful in reducing the dimensionality of the data to be processed by the classifier: reducing the dimensionality of the data reduces the sizes of the hypothesis space and, thus, results in faster execution time and improving predictive accuracy (inclusion of irrelevant features can introduce noise into the data, thus obscuring relevant features). Therefore, in our context of metric-based software quality classification, we could need a subset of metrics that can discriminate between the non-defect-prone and defect-prone classes. However, our approach does not need any feature selection, because its classification algorithm uses data reduction: deletion of irrelevant data (antigens) during the generation of B-cells.

	Numbers of				
	Classes	KLOCs	Blobs	FDs	SCs
Gantt Project	188	31	4	4	4
XercesJ	589	240	15	15	18
Total	777	271	19	19	22

Table 5.2 – System characteristics.

5.2 Study Definition and Design

We perform a series of experiments to assess the performance in time, precision, and recall of our novel approach for design-defects detection. Following the Goal Question Metric (GQM) methodology [10], the *goal* of our experiments is to analyse the performance of our approach and understand whether it performs better than previous approaches. The *purpose* is to provide an approach for design-defects detection. The *quality focus* is to provide a set of defect occurrences (*i.e.*, classes with characteristics violating design principles) with good precision and recall and in a reasonable time. The *perspective* is both of developers and quality assurance personnel, who perform evaluation activities and are interested in locating accurately parts of a system that need improvements; and researchers, who want to study design defects. The *context* of our experiments is two open-source Java systems: GanttProject v1.10.2 and XercesJ v2.7.0.

5.2.1 Objects

We conduct our experiments using two open-source Java systems: GanttProject v1.10.2 and XercesJ v2.7.0, which characteristics are summarised in Table 5.2. GanttProject[2] is a system for creating project schedules by means of Gantt charts and resource-load charts. It enables breaking down projects into tasks and establishing dependencies among tasks. XercesJ[3] is a family of packages for parsing and manipulating XML files. It implements a number of standard API for XML parsing, including DOM, SAX, and SAX2. We chose these systems because they are medium-size systems and manually-validated occurrences of Blob, Functional Decomposition (FD), and Spaghetti Code (SC) are available [70, 90].

[2]`http://ganttproject.biz/index.php`
[3]`http://XercesJ.apache.org/`

5.2.2 Research Questions

We want to answer the two research questions:

- **RQ1: To what extent an AIS-based approach can detect design defects in a system?**
- **RQ2: Is our approach better than state of-the-art approaches, such as DECOR, and BBNs?**

We answer **RQ1** in the following two scenarios:

- *intra-system identification*: In this first scenario, we study how knowledge of previously-detected Blobs in a given system, XercesJ v2.7.0, can help predict occurrences of other design defects in the same system. We divide the classes of XercesJ in three subsets with 16 occurrences of Blobs in each subset. Then, we train IDS on two of the subsets and apply it on the third subset (of the same system) in a 3-fold cross-validation.
- *extra-system identification*: In this second scenario, we study the performance of our approach using heterogeneous data. We assume that a developer has access to historical data from one system, *eg.*, GanttProject. We use this data to detect occurrences of design defects in the other system, XercesJ. We also perform the same study in the other direction, *i.e.*, using design defects in XercesJ to detect occurrences in GanttProject.

We answer **RQ2** by comparing the performance of IDS in precision, and recall, as computed in Scenario 2, against that of previous approaches.

In each scenario and research question, we use publicly-available data [70, 90] as oracle. We collect the numbers of true and false positive occurrences of the design defects detected by our approach and compare them with the oracle using the following IR metrics defined in [8]:

$$precision = \frac{|correct \cap detectedd|}{|detected|}$$

$$recall = \frac{|correct \cap detected|}{|correct|}$$

	Numbers of			
	Design Defects	False Positives	Precisions	Recalls
Subset 1	16	1	94.11%	100%
Subset 2	16	2	88.23%	100%
Subset 3	16	2	88.23%	100%
Average			90.19%	100%

Table 5.3 – Intra-system detection on XercesJ: 3-fold cross validation.

where *correct* represents the set of known instances of the design defects and *detected* that of candidate occurrences detected by an approach.

5.3 Study Results, Analyses and Discussions

We now discuss the results of our experiments.

RQ1: To what extent an AIS-based approach can detect design defects in a system?
In this first scenario, we use 3-fold cross validation. Table 5.3 shows the precisions corresponding to each fold. The average precision is 90.19% and the recall is 100%. The results also confirmed that IDS is not limited to the detection of a specific design defect: although we train our approach on instances of Blob, FD, and SC, it was also able to detect LargeClass and LongMethod. Overall, IDS detects all defect classes, *i.e.*, classes deviating from specific *good* design rules, *exemplified* by some design defects.

In the second scenario, we trained our approach on GanttProject v1.10.2 and applied it on XercesJ v2.7.0 and vice-versa. Table 5.4 shows the results. On XercesJ, our approach achieved a precision above 80%. The precision for GanttProject, although slightly lower at 65.0%, is still interesting considering that the approach was trained on a system from a different context. Moreover, a review of the false positives show that these classes have characteristics similar to the defects and, therefore, may eventually degenerate in design defects in the near future. Hence, they are also interesting to developers as they may be interested in preventing their further decay. Our approach in both cases achieved 100% recall, succeeding in returning all defect classes in the systems.

	GanttProject (Trained on XercesJ' Blobs, FDs, SCs)	XercesJ (Trained on GanttProject's Blobs, FDs, SCs)
# of Classes	188	589
# of Design Defects Instances	20	54
# of False Positives	7	10
Precision	65.0%	81.48%
Recall	100%	100%

Table 5.4 – Inter-system detection.

	GanttProject				XercesJ			
	DECOR	BBNs	IDS		DECOR	BBNs	IDS	
			Group 1	Group 2			Group 1	Group 2
Blob	16 (8.5%)	7 (3.7%)	34		44 (7.6%)	41 (6.9%)	55	
	4 (2.1%)	4 (2.1%)	4	20 (10.6)	15 (2.5%)	15 (2.5%)	15	54 (9.1%)
	25 %	57.1 %	11.76%	13 (6.9%)	33.3%	36.5 %	27.27%	44 (7.4%)
FD	15 (8.0%)	–	–	65%	29 (5.6%)	–	–	81.48%
	4 (2.1%)	–	–		15 (2.9%)	–	–	
	26.7%	–	–		51.7%	–	–	
SC	14 (7.4%)	–	–		76 (14.8%)	–	–	
	4 (2.1%)	–	11		18 (3.0%)	–	18	
	28.5%	–	32.35%		23.68%	–	32.72%	
Average	26.73%	57.1 %		65%	36.22%	36.5%		81.48%

Table 5.5 – Results of applying the detection approaches. Group 1 represents the training on Blob. Group 2 represents the training on Blob, FD, SC. (In each row, the first line is the number of detected classes, the second is the number of classes *being* design defects, the third is the precision. Numbers in parentheses are the percentages of classes being reported)..

We thus answer **RQ1** positively: the results suggest that even in the absence of historical data on a specific system, developers or quality assurance personnel could use IDS on different systems and obtain good precision and recall.

RQ2: Is our approach better than state of-the-art approaches, such as DECOR, and BBNs?

To answer RQ2, we compare the results of our approach against that of the state-of-the–art approaches: DECOR [90] and BBNs [70]. Table 5.5 summarises the results achieved by each approach. Globally, IDS outperforms DECOR and BBNs in term of precision. Moreover, DECOR and BBNs require expensive tuning by experts (in time and knowledge) to have acceptable precision and recall. Indeed, DECOR relies on rule cards built by expert while BBNs need experts' knowledge to build their learning structure. For these two approaches, an incomplete experts' knowledge can cause a high number of false positives, resulting in a waste of time and resources for developers that must skim through the results. IDS does not rely on any experts' knowledge but on a set of metrics characterising known instances of defects. Therefore, IDS reduces the bias introduced in BBNs by the experts structuring the BBNs and in DECOR when crafting rule cards.

Moreover, contrary to DECOR and BBNs, IDS detects a larger set of design defects. As presented in Table 5.5, when trained only on instances of Blob, IDS was *still* able to detect the instances of the other defects: on GanttProject, it returned all the 4 true occurrences of Blob and also 11 true occurrences of SC; on XercesJ, it returned again all the 15 true occurrences of Blob, 18 true occurrences of SC, and 14 occurrences of LargeClass. IDS also reported classes with borderline structure that may decay in design defects in the near future. Another strong point of IDS is its computation time, *i.e.*, in the order of milliseconds, *eg.*, on XercesJ, IDS detects Blob occurrences in 0.26s, while DECOR takes 2.4s.

We thus answer **RQ2** also positively: our approach has precisions and recalls superior than those of DECOR and BBNs and can detect similar defects as well as classes similar but not identical to some design defects.

5.4 Threats to Validity

The main threat to the *external validity* of our experiments that could affect the generalisation of the presented results relates to the analysed systems. We only used two medium-size systems, yet they support different activities and are open-source, thus available for replication. We plan to replicate our experiments on larger systems to confirm our results.

The subjective nature of specifying and detecting design defects and assessing detected classes is a threat to the *internal validity* of our experiments. Our understanding of design defects may differ from that of others. Our oracle, used to analyse our approach, was manually built by analysing the two systems used in the experiments. Three of the authors independently re-validated the publicly-available data [70, 90] to reduce the risk of classification errors. Finally, a candidate occurrence was classified as a real defect only when two out of three authors classified it as such. Such a process makes us quite confident about the accuracy of the oracle.

5.5 Summary

The detection of design defects in object-oriented software systems is important to improve and assess the quality of the systems, to ease their maintenance and evolution, and, thus, to reduce the overall cost of their development and ownership.

Current automated detection approaches are difficult to develop and put into place because they require experts' knowledge and interpretation. Moreover, they focus on detecting one kind of design defects at a time, while some defects are similar and classes with characteristics **similar but not identical** to some defects are also of interest to developers and quality assurance personnel.

In this chapter, we presented the first systematic parallel between artificial immune systems and the detection of design defects; a machine learning technique inspired from the immune system of the human body.

We performed experiments using GanttProject v1.10.2 and XercesJ v2.7.0 and the Blob, FD, and SC design defects. The experiments showed that an AIS can detect defects in systems with good precision and recall and address the limitations of previous work: it does not require experts' knowledge and interpretation and it can report classes that are similar but not identical to the detected defect.

Moreover, our AIS-based approach has the following additional benefits with respect to previous approaches. **Generalisation**: It does not need all of the data set to detect similar or identical occurrences of the design defects. It has data reduction capability: it does not require feature selection, *i.e.*, choosing the set of metrics. **Parameter Stability**: Current freely-available implementation of AIS are not optimised for the detection of design defects but already provide good precision and recall. **Adaptability**: It is adaptable and, in some cases, self-organising and thus can automatically identify new patterns in the data to create a different representation of the data being learnt. **Portability across Systems**: It has a good precision and recall when applied to different systems while previous work require recalibration of the conditional probabilities [70] or changes of thresholds [90, 108]. **Simplicity and Self-regulatory**: It does not require a topology or a rule card: no experts' knowledge and interpretation. Thus, it does not embed the experts' subjective understanding of the design defects. However, it requires an oracle providing occurrences of some defects. **Performance**: It has good performance in terms of precision, and recall, and it computation is very fast. The results of the experiments showed that its precision and recall are comparable or superior to that of previous approaches.

We conclude that the application of an artificial immune system to detect design defects is valuable. The immune system provides an interesting metaphor for detecting design defects.

CHAPTER 6

CONCLUSION

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

6.1 Dissertation Findings and Conclusions

Software systems undergo changes throughout their lifetimes as new features are added and bugs are fixed [81, 82]. As these systems evolved, their designs tend to decay with time and become harder to maintain [28, 122].

Design decay is the deviation of actual software design from the original one, *i.e.*, the violation of design choices during evolution [63, 100, 127]. Design decay occurs when changes are made on a system by developers who do not understand its original design [99]. On the one hand, making software changes without understanding their effects may lead to the introduction of bugs and the premature retirement of the system. On the other hand, when developers lack knowledge and–or experience in solving a design problem, they may introduce design defects. Therefore, developers need mechanisms to quantify design decay, to understand how a change to a system will impact the rest of the system, and tools to detect design defects. In this dissertation, we proposed to address three main problems: design decay evaluation, change impact analysis, and design defects detection.

In our *first contribution* [58, 59], we proposed a novel approach, called $ADvISE$, that exploits a set of metrics to measure design decay: the *Tunnel Triplets Metric* (TTM) and *Common Triplets Metric* (CTM). These metrics could be used as predictors of bug proneness [123] and design defect proneness [85]. We show that our design-decay metrics (TTM and CTM) provide us useful insights regarding the signs of design decay. If TTM decreases, then the original design decays. If TTM is stable, then the original design is stable, which means that the system is more adapted to the new changing requirements. If CTM increases, then new requirements and new functionalities are implemented in the system. If CTM is stable, then the system is stable and the most of maintenance activities are bug fixes. As shown by our experiments, design decay is inevitable, but it is possible to evaluate it.

In our *seconde contribution* [57], we proposed a novel approach to change impact analysis specifically designed to study the scope of change propagation. We showed that changes propagate in systems, like earthquakes. The change impact is most severe near the epicenter class and drops off away for the classes in higher levels. Identifying the scope of change propagation could help, both developers and managers. Developers could locate easily the change impact and, thus, they do not have to analyse the whole source code to understand the ripple effect of a change. Managers could estimate the efforts required to perform changes more accurately.

In our *third contribution* [56], we proposed a novel approach for design defects detection, called IDS (Immune-based Detection Strategy), based on Artificial Immune Systems. We evaluated our approach on finding potential defects (Blob, FD, and SC) in two open-source systems (GanttProject and XercesJ) and showed that an AIS can detect defects in systems with good precision and recall and address the limitations of previous work: it does not require experts' knowledge and interpretation and it can report classes that are similar but not identical to the detected defects. We conclude that the application of an artificial immune system to detect design defects is valuable. The immune system provides an interesting metaphor for detecting design defects.

We found external information in bug reports and mailing lists that confirm that our approaches are able to evaluate design decay and identify the impact of changes. Also, we showed that our design-defects detection approach has a good precision and recall. Therefore, it is possible to provide solutions that help developers in evaluating design decay, analysing the change impact and detecting design defects, and thus we confirm our thesis.

6.2 Opportunities for Future Research

We could explore different future work directions. Following our *first contribution*, we plan to investigate the use of other metrics: the first measure of design decay is related to how many of the classes of a considered design are kept in subsequent versions or releases. Such measure, although unidimensional, is simple, intuitive, and can be used for more complex notions, such as estimating a "mortality" rate for classes in a system. The second measure considers the number of connected components (micro-designs) of a design in subsequent releases. Considering the set of the classes of a given design, it may happen that the overall connectivity is not preserved. By deleting some relations (*eg.*, when trying to insert some new intermediary classes), one may add a degree of separation between previously connected classes.

In the *second contribution*, we plan to adapt seismology models to predict changes to classes. In the case of earthquakes, seismologists are interested in debris forecasting, to predict the quantity of damage and seek to minimise the earthquake impact through the improvement of construction standards. Earthquake prediction technique generally use probabilistic methods to predict earthquake risk using past history. We will study the possibility of using the data about previous changes to forecast "earthquakes" that could occurs in a system, their potential damages, and the factors influencing their propagations.

In the *third contribution*, we plan to compare our approach with other machine learning techniques, such as support vector machine, and to further study the parameters of the approach, including refining the choice of characteristics of classes. We also plan to extend our metaphor to other problems, such as the prediction of "buggy" changes, when modifying a class. The idea is to classify the changes as clean or buggy. After training an AIS by using change data from revisions 1 to n, if there is a new and unclassified change, *i.e.*, revision $n + 1$, this change can be classified as either buggy or clean by using the trained classifier model. In this manner, the change classification predicts whether a new change is more similar to prior "buggy" or clean changes.

In this dissertation, we highlighted that the design quality of a software system deteriorates throughout its evolution due to design decay. Therefore, we could propose to improve the design quality of a system by identifying refactoring opportunities, which resolve design defects existing in source code. Thus, it could possible to provide solutions that help developers in improving design quality by appropriate refactorings.

RELATED PUBLICATIONS

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

Journal articles

1. **Salima Hassaine**, Yann-Gäel Guéhéneuc, Sylvie Hamel, Bram Adams, and Giulio Antoniol (2012). *Evaluating Design Decay during Software Evolution*, Journal of Empirical Software Engineering (EMSE) (submitted).

Conference articles

1. **Salima Hassaine**, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giulio Antoniol (2012). *ADvISE: Architectural Decay In Software Evolution*. In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), March 27-30, 2012, Szeged, Hungary. IEEE Computer Society Press.

2. **Salima Hassaine**, Ferdaous Boughanmi, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giulio Antoniol (2011). *A Seismology-inspired Approach for Change Impact Analysis*. In Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), September 25 - 30, 2011, Williamsburg, VA, USA. IEEE Computer Society Press.

3. **Salima Hassaine**, Ferdaous Boughanmi, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giuliano Antoniol (2011). *Change Impact Analysis : An earthquake Metaphor*. In Proceedings of the 19th International Conference on Program Comprehension (ICPC), June 22 - 24, 2011, Kingston, Ontario, Canada. IEEE Computer Society Press.

4. **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.

BIBLIOGRAPHY

[1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 84, 1990.

[2] Ieee standard for software maintenance. *IEEE Std. 1219-1998*, page 52, 1998.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM, 1990. ISBN 0-89791-364-7.

[4] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Softw. Pract. Exper.*, 31(4):331–355, 2001. ISSN 0038-0644.

[5] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. *Principles of Software Evolution, International Workshop on*, 0:31–40, 2004. ISSN 1550-4077.

[6] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi. Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories. In *the 2005 international workshop on Mining software repositories*, pages 1–5. ACM, 2005. ISBN 1-59593-123-6.

[7] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *the Conference on Software Maintenance*, pages 292–301. IEEE Computer Society, 1993.

[8] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[9] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28: 4–17, January 2002.

[10] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng.*, 10(6):728–738, 1984.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-19930-0.

[12] Douglas Bell. *Software Engineering, A Programming Approach*. Addison-Wesley, 2000.

[13] Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1): 53–65, 2002.

[14] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 9: one-way analysis of variance. *Critical care*, 8(2):130–136, 2004. ISSN 1466-609X.

[15] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelCase or Under_score. In *Proceedings of International Conference on Program Comprehension*, pages 158–167. IEEE Computer Society Press, 2009.

[16] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *the International Conference on Software Maintenance*, pages 44–54. IEEE Computer Society, 2003. ISBN 0-7695-1905-9.

[17] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[18] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In Peter Kokol, editor, *IASTED Conference on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[20] Jan Bosch. Evolution and composition of reusable assets in product-line architectures: A case study. In *Proceedings of the TC2 First Working IFIP*

Conference on Software Architecture (WICSA1), pages 321–340, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[21] Salah Bouktif, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *the 13th Working Conference on Reverse Engineering*, pages 221–230, 2006.

[22] Ghizlane El boussaidi. *Développement logiciel par transformation de modèles.* PhD thesis, Université de Montréal, 2010.

[23] 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, 1998.

[24] Jason Brownlee. Artificial immune recognition system: a review and analysis. Technical Report 1-02, Swinburne University of Technology, 2005.

[25] Jason Brownlee. Clonal selection theory clonalg. the clonal selection classification algorithm. Technical Report 2-02, Swinburne University of Technology, 2005.

[26] K. E. Bullen. *An introduction to the theory of seismology*. Cambridge University Press, Cambridge, 3rd ed. edition, 1963.

[27] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, pages 29–. IEEE Computer Society, 2005.

[28] S. Jeromy Carrière and Rick Kazman. The perils of reconstructing architectures. In *Proceedings of the third international workshop on Software architecture*, pages 13–16, New York, NY, USA, 1998. ACM. ISBN 1-58113-081-3.

[29] Jerome H. Carter. The immune system as a model for pattern recognition and classification. *American Medical Informatics Association*, 7(1):28–41, 2000.

[30] Cagatay Catal and Banu Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences, Elsevier*, 179(8):1040–1058, 2009.

[31] Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 163–166. ACM, 2010. ISBN 978-1-60558-719-6.

[32] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions in Software Engineering*, 20(6):476–493, 1994.

[33] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977. ISBN 0-471-16240-X.

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

[35] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490. ACM, 2008.

[36] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34: 497–515, 2008.

[37] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1 –12, 2001.

[38] H.A. Eiselt, M. Gendreau, G. Laporte, and Université de Montréal. Centre de recherche sur les transports. *Arc Routing Problems: The Chinese postman problem*. Publication (Université de Montréal. Technical Report CRT-960, Centre de recherche sur les transports). Université de Montréal, Centre de recherche sur les transports, 1993.

[39] Laleh M. Eshkevari, Venera Arnaoudova, Massimiliano Di Penta, Rocco Oliveto, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of identifier renamings. In *Proceeding of the 8th working conference on Mining software repositories*, pages 33–42. ACM, 2011.

[40] Martin Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1^{st} edition, June 1999.

[41] William B. Frakes and Ricardo A. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, 1992. ISBN 0-13-463837-9.

[42] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–. IEEE Computer Society, 1998.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1^{st} edition, 1994.

[44] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21., Carnegie Mellon University, Software Engineering Institute, 1994.

[45] Daniel M. German, Ahmed E. Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology.*, 51:1394–1408, 2009. ISSN 0950-5849.

[46] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In *In Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 40–49. IEEE Computer Society, 2004.

[47] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00)*, 2000.

[48] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[49] C.W.J Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–38, 1969.

[50] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1st *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, 2005.

[51] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the* 19th *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–314. ACM Press, 2004. 14 pages.

[52] Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE)*, 34(5):667–684, 2008. 18 pages.

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

[54] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Khashayar Khosravi, and Houari Sahraoui. Design patterns as laws of quality. University of Montreal, 2005. URL `http://www.iro.umontreal.ca/~ptidej/Publications/Documents/OODK05.doc.pdf`.

[55] Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison-Wesley, 2001.

[56] Salima Hassaine, Foutse Khomh, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Ids: An immune-inspired approach for the detection of software design smells. In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, QUATIC '10, pages 343–348. IEEE Computer Society, 2010.

[57] Salima Hassaine, Ferdaous Boughanmi, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. A seismology-inspired approach to study change propagation. In *Proceedings of the 2011 27th IEEE International Conference on*

Software Maintenance, pages 53–62. IEEE Computer Society, 2011. ISBN 978-1-4577-0663-9.

[58] Salima Hassaine, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. Advise: Architectural decay in software evolution. In *Proceedings of the* 16^{th} *European Conference on Software Maintenance and Reengineering*, pages 267–276. IEEE Computer Society, 2012.

[59] Salima Hassaine, Fahmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Bram Adams. Evaluating design decay during software evolution. *Empirical Software Engineering (submitted)*, 2012.

[60] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *the 20th IEEE International Conference on Software Maintenance*, pages 284–293. IEEE Computer Society, 2004.

[61] Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11:335–367, 2006. ISSN 1382-3256.

[62] Frederick S. Hillier, Gerald J. Lieberman, Frederick Hillier, and Gerald Lieberman. *Introduction to Operations Research.* McGraw-Hill, 2004.

[63] Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration: a survey. *Information Software Technology*, 47:643–656, 2005. ISSN 0950-5849.

[64] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-32571-3.

[65] Mehdi Jazayeri. On architectural stability and evolution. In *da-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 13–23, London, UK, 2002. Springer-Verlag. ISBN 3-540-43784-3.

[66] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. *csmr*, 0:175–184, 2006.

ISSN 1052-8725. doi: http://doi.ieeecomputersociety.org/10.1109/CSMR.2006.25.

[67] Y. Y. KAGAN and L. KNOPOFF. Statistical short-term earthquake prediction. *Science*, 236(4808):1563–1567, 1987.

[68] Marouane Kessentini, Stephane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the 25th International Conference on Automated Software Engineering*. IEEE Computer Society Press, September 2010.

[69] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE Computer Society, 2009.

[70] 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, 2009. 10 pages.

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

[72] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

[73] D. Kimelman, M. Kimelman, D. Mandelin, and D.M. Yellin. Bayesian approaches to matching architectural diagrams. *Software Engineering, IEEE Transactions on*, 36(2):248 –274, 2010.

[74] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of ACM*, 46:604–632, 1999.

[75] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, and Giuliano Antoniol. Recovering the evolution stable part using an ecgm algorithm: Is there a tunnel in mozilla? In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 179–188, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3589-0. doi: http://dx.doi.org/10.1109/CSMR.2009.24.

[76] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[77] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1):141–175, 2011.

[78] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[79] James R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, 1999. ISSN 0362-1340.

[80] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[81] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X.

[82] Meir M. Lehman, Juan F. Ramil, P. D. Wernick, Dewayne E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the IEEE Symposium on Software Metrics*, pages 20–32. IEEE Computer Society, 1997.

[83] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966.

[84] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007.

[85] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. *Software Maintenance and Reengineering, European Conference on*, 0:277–286, 2012.

[86] Haroon Malik and Ahmed E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In *ICSM'08*, pages 177–186, 2008.

[87] Mika Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study.* PhD thesis, Helsinki University of Technology, 2003.

[88] 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.

[89] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 177–188, June 2007.

[90] 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. *Transactions on Software Engineering (TSE)*, 2009. 16 pages.

[91] A. Morales-Esteban, F. Martínez-Álvarez, A. Troncoso, J. L. Justo, and C. Rubio-Escudero. Pattern recognition to forecast seismic time series. *Expert System Application*, 37:8333–8342, 2010. ISSN 0957-4174.

[92] Robert Moreton. A process model for software maintenance. *Journal of Information Technology (Routledge, Ltd.)*, 5(2):100–104, 1990.

[93] 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, 2005.

[94] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the* 3^{rd} *ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM, 1995.

[95] Stephen C. Myers and William R. Walter. Using epicenter location to differentiate events from natural background seismicity. Technical Report, it was prepared for submittal to the 21 st Seismic Research Symposium: Technologies for Monitoring the Comprehensive Nuclear-Test-Ban Treaty UCRL-JC-134301; GC0402000, Lawrence Livermore National Laboratory, 1999.

[96] Terri Oda and Tony White. Increasing the accuracy of a spam-detecting artificial immune system. In *Proceedings of the Congress on Evolutionary Computation (CEC 2003)*, volume 1, page 390396, Canberra, Australia, 2003.

[97] 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 *Proceedings of the* 3^{rd} *International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.

[98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[99] David Lorge Parnas. Software aging. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[100] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, 1992. ISSN 0163-5948.

[101] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice.* Prentice-Hall, 1998.

[102] S.L. Pfleeger and S.A. Bohner. A framework for software maintenance metrics. In *Proceedings of the* 6th *International Conference on Software Maintenance*, pages 320 –327, 1990.

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

[104] Roger S. Pressman. *Software Engineering – A Practitioner's Approach.* McGraw-Hill Higher Education, 5th edition, November 2001.

[105] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31:2005, 2005.

[106] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Softw.*, 21:62–69, July 2004.

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

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

[109] A. Saichev and D. Sornette. Theory of earthquake recurrence times. *J.GEOPHYS.RES.*, 112:B04313, 2007.

[110] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical Report GIT-CERCS-10-10, Georgia Institute of Technology. Center for Experimental Research in Computer Systems, 2011.

[111] Bordin Sapsomboon. *Shared Defect Detection : The Effects of Annotations in Asynchronous Software Inspection.* PhD thesis, University of Pittsburgh, 2000.

[112] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th inter-*

national conference on Software engineering, pages 471–480, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

[113] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007. ISBN 1584888148, 9781584888147.

[114] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR'01)*. IEEE CS Press, 2001.

[115] Ian Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2000.

[116] Mikael Svahnberg and Jan Bosch. Characterizing evolution in product-line architectures. In *In Proceedings of the IASTED* 3^{rd} *International Conference on Software Engineering and Applications*, pages 92–97, 1999.

[117] Burton E. Swanson. The dimensions of maintenance. In *Intl. Conf. on Software Engineering*, pages 492–497, San Francisco, California, 1976. IEEE Computer Society.

[118] Armstrong A. Takang and Penny A. Grubb. *Software maintenance: concepts and practice*. International Thomson Computer Press, UK, 1996.

[119] Jon Timmis and Thomas Knight. Artificial immune systems: Using the immune system as inspiration for data mining, 2001.

[120] 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.

[121] 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, 2002.

[122] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[123] Jilles van Gurp, Sjaak Brinkkemper, and Jan Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *Journal of Software Maintenance and Evolution*, 17:277–306, 2005. ISSN 1532-060X.

[124] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[125] Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1^{st} edition, 1995.

[126] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25: 446–452, 1982. ISSN 0001-0782.

[127] B.J. Williams and J.C. Carver. Characterizing software architecture changes: An initial study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 410 –419, 2007.

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

[129] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 325–334. ACM, 2010.

[130] Zhenchang Xing. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Trans. Softw. Eng.*, 31(10):850–868, 2005. ISSN 0098-5589. Member-Stroulia, Eleni.

[131] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 34 – 43, june 2004.

[132] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM inter-*

national Conference on Automated software engineering, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4.

[133] Zhenchang Xing and Eleni Stroulia. API-evolution support with diff-CatchUp. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 33 (12):818 – 836, 2007.

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

[135] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *the 2005 international workshop on Mining software repositories*, pages 1–5. ACM, 2005.

[136] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *In Proceedings of the Conference on Software Maintenance and Reengineering*, pages 134–142. IEEE Computer Society, 2005.

[137] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12:143–160, 2007. ISSN 1382-3256.

[138] Yu Zhou, Michael Würsch, Emanuel Giger, Harald C. Gall, and Jian Lü. A bayesian network based approach for change coupling prediction. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE '08, pages 27–36. IEEE Computer Society, 2008.

[139] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

APPENDIX A

DEFINITIONS OF METRICS AND QUALITY ATTRIBUTES

This Appendix presents the definitions of the quality attributes [9, 43, 54], and 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.

CoAttributes: 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 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 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.

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.

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.

A.2 Software Quality Attributes

- 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.

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.

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 system.

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 system.

Blob: (called also God class [107]) 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 [128].

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 [40].

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.