Université de Montréal

Analysing Artefacts Dependencies to Evolving Software Systems

par
Fehmi Jaafar

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, 2013

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Analysing Artefacts Dependencies to Evolving Software Systems

présentée par :

Fehmi Jaafar

a été évaluée par un jury composé des personnes suivantes :

Président-rapporteur	:	Nadia El-Mabrouk
Directeur de recherche	:	Yann-Gaël Guéhéneuc
Codirectrice	:	Sylvie Hamel
Membre du jury	:	Giovanni Beltrame
Examinateur externe	:	Martin Pinzger
Représentant du doyen de la FAS	:	Nadia El-Mabrouk

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. Lorsque les logiciels évoluent, leurs architectures ont tendance à se dégrader et deviennent moins adaptables aux nouvelles spécifications des utilisateurs. En effet, les architectures de ces logiciels deviennent plus complexes et plus difficiles à maintenir à cause des nombreuses dépendences entre les artefacts. Par conséquent, les développeurs doivent comprendre les dépendences entre les artefacts des logiciels pour prendre des mesures proactives qui facilitent les futurs changements et ralentissent la dégradation des architectures des logiciels.

D'une part, le maintien d'un logiciel sans la compréhension des les dépendences entre ses artefacts peut conduire à l'introduction de défauts. D'autre part, lorsque les développeurs manquent de connaissances sur l'impact de leurs activités de maintenance, ils peuvent introduire des défauts de conception, qui ont un impact négatif sur l'évolution du logiciel. Ainsi, les développeurs ont besoin de mécanismes pour comprendre comment le changement d'un artefact impacte le reste du logiciel.

Dans cette thèse, nous proposons trois contributions principales :

- La spécification de deux nouveaux patrons de changement et leurs utilisations pour fournir aux développeurs des informations utiles concernant les dépendences de co-changement.
- La spécification de la relation entre les patrons d'évolutions des artefacts et les fautes.
- La découverte de la relation entre les dépendances des anti-patrons et la prédisposition des différentes composantes d'un logiciel aux fautes.

ABSTRACT

Program maintenance accounts for the largest part of the costs of any program. During maintenance activities, developers implement changes (sometimes simultaneously) on artefacts to fix bugs and to implement new requirements. Thus, developers need knowledge to identify hidden dependencies among programs artefacts and detect correlated artefacts.

As programs evolved, their designs become more complex over time and harder to change. In the absence of the necessary knowledge on artefacts dependencies, developers could introduce design defects and faults that causes development and maintenance costs to rise. Therefore, developers must understand the dependencies among program artefacts and take proactive steps to facilitate future changes and minimize fault proneness. On the one hand, maintaining a program without understanding the different dependencies between their artefacts may lead to the introduction of faults. On the other hand, when developers lack knowledge about the impact of their maintenance activities, they may introduce design defects, which have a negative impact on program evolution. Thus, developers need mechanisms to understand how a change to an artefact will impact the rest of the programs artefacts and tools to detect design defects impact.

In this thesis, we propose three principal contributions. The *first contribution* is two novel change patterns to model new co-change and change propagation scenarios. We introduce the Asynchrony change pattern, corresponding to macro co-changes, *i.e.*, of files that co-change within a large time interval (change periods), and the Dephase change pattern, corresponding to dephase macro co-changes, *i.e.*, macro co-changes that always happen with the same shifts in time. We present our approach, named Macocha, and we show that such new change patterns provide interesting information to developers.

The *second contribution* is proposing a novel approach to analyse the evolution of different classes in object-oriented programs and to link different evolution behavior to faults. In particular, we define an evolution model for each class to study the evolution and the co-evolution dependencies among classes and to relate such dependencies with fault-proneness.

The *third contribution* concerns design defect dependencies impact. We propose a study to mine the link between design defect dependencies, such as co-change dependencies and static relationships, and fault proneness. We found that the negative impact of design defects propagate through their dependencies.

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

- The new change patterns, Asynchrony and Dephase change patterns, provide developers useful insights regarding hidden co-change dependencies and improve precision and recall of co-change analysis.
- Two major kinds of class evolution dependencies, *i.e.,* class lifetime and class co-evolution, inform us about fault-proneness. Indeed, Persistent classes are significantly less fault-prone than other classes and faults fixed to maintain co-evolved classes are significantly more frequent than faults fixed using not co-evolved classes.
- Dependencies with anti-patterns have a negative impact on fault-proneness. We show that classes having static relationships with anti-patterns or co-changing with anti-patterns are significantly more fault-prone than other classes.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

Macocha	Macro Co-change Identification
OO	Object Oriented
MCC	Macro co-change
DMCC	Dephase Macro co-change
S_{MCC}	The set of macro co-changing files with identical profiles in a program
S_{DMCC}	The set of dephase macro co-changing files identified when shifting profiles
S_{MCCH}	The set of approximate macro co-changing files with similar profiles in a progran
S_{DMCCH}	The set of approximate dephase macro co-changing files identified by using the H
S_{AR}	The set of co-changed files found by an approach based on Association Rules
Profilo	Evolution Profile detection and analysis
ADvISE	Architectural Decay In Software Evolution
CVS	Concurrent Versions System
DECOR	Defect dEtection for CORrection
KNN	k-Nearest Neighbor Algorithm
PADL	Pattern and Abstract-level Description Language
POM	Primitives, Operations, Metrics
Ptidej	Pattern Trace Identification, Detection, and Enhancement in Java
SVN	Apache SubVersioN
UML	Unified Modeling Language

To my parents

To my wife

To my brothers and sisters

To my friends

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 thesis.

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

Great thanks to my fellow doctoral students from the Ptidej and Soccer labs for creating an enjoyable working environment. Special thanks go to Salima Hassaine and Darine Ameyed. 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 gratitude to my parents Amor Jaafar and Miriam Elmonsor, and my wife Wided, 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.

This research was partially supported by FQRNT, NSERC, the Research Chairs in Software Patterns and Patterns of Software and the Tunisian Ministry of Higher Education and Scientific Research.

— Fehmi Jaafar

CHAPTER 1

INTRODUCTION

1.1 Research Context

Programs evolve continuously, requiring constant maintenance and development [66]. Maintenance has been recognized as the most costly and difficult phase in the software life cycle [10, 94]. Maintenance effort has been estimated to be frequently more than 70% of the overall software development cost [81]; an increase due in part to *design decay*. Design decay is the deviation of an actual software design from its originally planned design, *i.e.*, the violation of design choices during evolution [45, 79, 103]. This design decay can be detected by measuring the instability of the program artefacts[1] [40], high fault rates [98], and poor code quality [26, 98].

For example, Ostrand *et al.* [77] found that 20% of artefacts contains 80% of faults. At the same time, these 20% of artefacts accounted for 50% of the source code. Assuming that all artefacts are considered to have the same likelihood for fault-proneness is not realistic, because, for example, not all artefacts are there to last forever, some are meant for experimentation, so it could be expected that they have more faults.

Developers must continually adapt programs to meet new requirements and user needs. Thus, they need knowledge to identify hidden dependencies among programs artefacts and to detect critical parts of source code. This knowledge will help to improve the speed [115] and accuracy of maintenance tasks [87] while reducing the risk of introducing faults in the source code.

At a low level, detecting and understanding dependencies among different artefacts is important from the points of views of both researchers and programmers. For researchers, it gives useful knowledge about the impact of such dependencies on program quality. For programmers, it provides knowledge concerning co-evolved classes: clusters of classes exhibiting similar evolution profiles, due to hidden inter-dependencies among them, not necessarily visible when examining their relations

[1] Artefacts can be source code files, classes in object-oriented programs, specifications...

at the code level [106]. It could help them to realise their maintenance tasks more simply and systematically [38]. At a high level, studying the dependencies between some motifs, such as anti-patterns, and the rest of the program artefacts, provide programmers with a simplified but accurate picture of the programs in time to understand their full complexity.

Previous literature describes several approaches to extract and analyse dependencies among artefacts and to infer the patterns that describe their changes [9, 53]. Several of these approaches identify co-changes among artefacts, *e.g.* [111, 115], which represent the (often implicit) dependencies or logical couplings among artefacts that have been observed to frequently change together [31]. Ying *et al.* [111] and Zimmermann *et al.* [115] applied association rules to identify co-changing files. Their hypothesis is that past co-changed files can be used to recommend source code files potentially relevant to a change request. An association-rule algorithm extracts frequently co-changing files in a transaction and categorizes them into sets that are regarded as change patterns to guide future changes. Such algorithm uses co-change history and avoids the source code dependency parsing process. However, it only computes the frequency of co-changed files in the past and omits many other cases, *e.g.* files that co-changed with always the same period of time between changes. In Chapter 3, we show that theses approaches miss interesting occurrences of co-changes, so called macro co-change because, by their very definition, they do not integrate the analysis of artefacts that are maintained by different developers and–or with some shifts in time.

In addition, artefacts that exhibit similar evolution profiles, due to interdependencies among them, are considered as co-evolved artefacts [106]. Thus, co-change is only one aspect of co-evolution. Indeed, if two artefacts co-changed then they co-evolved. But if two artefacts co-evolved, they not necessarily co-changed. Xing *et al.* [106] analysed the evolution profiles of classes. Class-evolution profiles report the complete history of changes made to an individual class in each subsequent version. Such approach identifies co-evolved classes but does not analyse the relations between class evolution profiles and architectural decay. Indeed, some of these classes are added, renamed, and changed in the same versions of the program. In Chapter 4, we found that these classes have similar evolution trends and that many of them are involved in the same faults. Indeed, we detect dependencies among

these classes and we analyse the impact of such dependencies on maintenance tasks such as change propagation.

Another factor affecting the effort required for maintenance is the design quality of programs [109]. 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 [28] and anti-patterns [17]. Design defects occur generally in object-oriented programs when developers lack knowledge and–or experience in solving a design problem or when applying some design patterns: "something that looks like a good idea, but which backfires badly when applied" [21]. They have a negative impact on the quality characteristics (*e.g.* change-proneness and fault-proneness [54, 95]) and the evolution of programs [17, 28]. Classes in anti-patterns have dependencies with other classes, such as static relationships, that may propagate potential problems to these other classes. In Chapter 5, we show that, in almost all releases of three programs, classes having dependencies with anti-patterns are more fault-prone than others. We also report other observations about these dependencies, such as their impact on fault prediction.

1.2 Problem Statement and Contributions

The above section leads us to formulate our thesis:

> ***Software maintenance and evolution is negatively impacted by uncontrolled changes, hidden artefacts dependencies, and design defects.***

To show our thesis, we propose to address three main problems: co-change analysis, evolution analysis, and design-defects dependencies analysis. To confirm our findings and to improve the precision and recall of used tools, we generally use external information, such as bug reports and mailing lists.

Contribution 1: Improvement of Co-change Analysis

Two artefacts are co-changing if they were changed by the same author and with the same log message in a time-window of less than 200 ms. [115]. Mockus *et al.* [72] defined the proximity in time of check-ins by the check-in time of adjacent files that differ by less than three minutes. Other studies (*e.g.* [27] and [33]) described issues about identifying atomic change sets and reported that, in all cases, they differed by few minutes.

Previous co-changes are intrinsically limited in time. They cannot express patterns of changes between long time intervals and–or committed by different developers. In Chapter 3 we present typical examples of co-changes not detected by any previous approach. We present, the first approach to detect and to report such co-changes.

We compare the results of our approach on different programs developed with three different programming languages, C, C++, and Java. We also use external information provided by bug reports, mailing lists, and requirement descriptions to validate co-changes not found using previous approaches and to show that they explain real evolution phenomena.

Therefore, our **first contribution** [47] is a novel approach, called *Macocha*, that introduces the novel concepts of *macro co-change* and *dephase macro co-change*, inspired from co-changes and using the concept of change periods. The *macro co-change* describes a set of files that always change together in the same periods of time. The *dephase macro co-change* describes a set of files that always change together with some shift in time in their periods of change.

We find that Macocha has a better precision and recall than association rules and can find novel occurrences of change patterns.

Contribution 2: Spotting the Relation Between Program Evolution and Fault-proneness

We find in the literature different approaches to analyse the evolution of software designs [56, 58, 106]. Most of these previous approaches aim at finding design changes and class evolution occurring in object-oriented programs. Identifying co-evolution dependencies is useful to study the relation between the evolution of

artefacts, their dependencies, and fault-proneness. However, existing approaches thus far did not link the evolution of classes to faults. Because (1) some classes evolve independently, other classes have to do it together with others (co-evolution); and (2) not all classes are meant to last forever, but some are meant for experimentation or to try out an idea that was then dropped or modified. Then, in our **second contribution** [48], we group classes based on their evolution to infer their lifetime models and co-evolution trends. Then, we link each group's evolution to faults.

We find that (1) classes with specific lifetime models are significantly less fault-prone than other classes and (2) faults fixed by maintaining co-evolved classes are significantly more frequent than faults fixed using other classes.

Contribution 3: Analysing the Impact of Anti-patterns Dependencies

Without proper knowledge, developers may introduce anti-patterns in programs. In theory, anti-patterns [102] are "poor" solutions to recurring problems. In practice, an anti-pattern is a literary form that describes a bad solution to recurring design problems that leads to negative effects on code quality [17].

Most previous work agree that anti-patterns render the maintenance of programs more difficult [1, 54]. However, there are only few previous work about the dependencies (static relationships and co-change dependencies) between anti-patterns and other artefacts in programs. Yet, understanding the impact of such dependencies, in particular on fault-proneness, help developers to better understand and maintain programs.

No empirical study has been conducted so far to detect and to analyse the fault proneness of classes that have static or co-change relationships with anti-patterns and to adjust the possibility of introducing faults during the maintenance of such classes.

Therefore, our **third contribution** [49] is the detection of the impact of design defects dependencies. We conduct an empirical study, performed on three object-oriented programs, which provides empirical evidence of the negative impact of

dependencies with anti-patterns on fault-proneness. Indeed, we find that having dependencies with anti-patterns can significantly increase fault-proneness.

1.3 Roadmap

The remainder of this thesis provides the following content: Chapter 2 reviews related work on evolution and co-evolution analysis, co-change analysis, and design defects impact. Chapter 3 reports our first contribution, improving co-change analysis. Chapter 4 reports our second contribution, studying the relation between the evolution and co-evolution of artefacts and fault-proneness.Chapter 5 reports our third contribution, which concerns anti-patterns dependencies impact on fault-proneness. Chapter 6 presents the conclusions of this thesis and outlines some directions of future research. Appendix A presents the definitions of metrics and quality attributes considered in this thesis. Appendix B presents the complete list of anti-patterns considered in this thesis with their definitions.

CHAPTER 2

RELATED WORK

This chapter provides a survey of work related to this thesis. The structure of the chapter is as follows: Section 2.1 recapitulates related work on software evolution. Section 2.2 provides a description of leading work on file stability. Section 2.3 discusses the state of the art on co-change dependencies analysis. Section 2.4 reviews earlier work on fault-proneness. Section 2.5 summarises exiting work on design defect detection and impact.

2.1 Software Design Evolution

Antoniol *et al.* [4] proposed an automatic approach, based on cosine similarity of class identifiers to automatically identify links between classes (obtained from refactoring) of 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, they applied a vector space model that treats these documents and queries as vectors [29]. The documents was ranked against queries to compute a similarity function between the corresponding vectors. Their approach does not take into account the relations between classes.

The phenomenon of software aging is the result of software evolution. Parnas [78] suggested that programs suffer from various aging problems such as increasing complexity, faults, unstructured code, feature overloading, etc. Eick *et al.* [26] suggested that a code is decayed if it is more difficult to maintain than it used to be. We believe, like the above cited authors, that code decay is essentially the result of program evolution. The design of a program deviates from its planned form with every new version of program to incorporate new features by implementing new classes and–or deleting, refactoring and changing old classes. We relate the evolution of classes in object-oriented programs and fault-proneness to emphasize program evolution consequences. Fraser [30] presented DiffTree to infer a phylo-

genetic tree from related programs. It described the retrospective computation of version trees for a set of programs, without mining source code control systems.

Gagandeep and Hardeep Singh [92] reported that that software tends to become more complex over the series of releases and maintaining them becomes a difficult task. The authors also investigated the applicability of Lehman's Law of Software Evolution using different metrics and found that Lehman's laws related to increasing complexity and continuous growth are supported by computed metrics.

DiffTree compared set of codes with one another, and presents a parsimonious phylogenetic tree for them. It can also help to identify cases where a repair made to one version was missed in others. We share with the author the idea that it is interesting to identify commonalities and divergences among versions to acknowledge the contributions of each version relative to one another. Karim *et al.* [52] described a method for constructing phylogeny models that used n-perms to match possibly permuted code and to discover malicious programs, such as viruses and worms, frequently related to previous programs through evolutionary relationships.

Demeyer *et al.* [25] presented an approach to understand how object-oriented programs have evolved by discovering which refactoring operations have been applied from one version of the software to the next.

Discussion

Previous approaches were useful to identify some replacement, merge and refactoring during the evolution of programs but they cannot relate program evolution with faults proneness.

Indeed, existing approaches for design evolution compare two versions of a software design to study its evolution. They adopted techniques to automatically identify evolution discontinuities when analyzing the evolution of object-oriented source code at class level. In this thesis, we will present approaches to analyse programs artefacts according to their evolutionary history to recognize and to understand co-evolution situations among them in the goal to relate these situations with faults.

2.2 File Stability

Many approaches exist to group files based on their relative stability throughout the software development life cycle. For example, UMLDiff [106] compares and detects the differences between the contents of two object-oriented program versions. A fact extractor parses each version to extract models of their design. Next, a differencing algorithm, UMLDiff, extracts the history of the program evolution, in terms of the additions, removals, moves, renamings, and signature-changes of design entities, such as packages, classes, interfaces, and their fields and methods. UMLDiff then assigns a stability to each class: short-lived classes (that exist only in a few versions of the program and then disappear), idle classes (that rarely undergo changes after their introduction in the program), and active classes (that keep being modified over their whole lifespan).

Kpodjedo *et al.* in [60] and [58] proposed to identify all files that do not change in the history of a program, using an Error Tolerant Graph Matching algorithm. They studied the evolution of the program class diagram by collecting program source code over several years, reverse-engineering their class diagrams, and recovering traceability links between subsequent class diagrams. Their approach identified evolving classes that maintain a stable structure of relations (association, inheritance, and aggregation) and so on, which likely constitute the stable backbone of a program.

Lanza *et al.* [62] presented an evolution matrix to display the evolution of the files of a program. Each column of the matrix represents a version of the program, while each row represents the different versions of the same file. Within the columns, the files are sorted alphabetically. Then, the authors presented a categorisation of files based on the visualisation of different versions of a file: a pulsar file grows and shrinks repeatedly during its lifetime, a supernova file suddenly explodes in size, a white dwarf is a file that used to be of a certain size, but lost its functionality, a red giant file tends to implement too many functionalities and is quite difficult to refactor, and an idle file does not change over several versions.

Discussion

The Error Tolerant Graph Matching algorithm and UMLDiff require parsing and comparing AST-like representations of the programs before performing their

analysis. We propose to compute stability in a more simple way by using the version control systems, wich keeps track of all work and all changes in each file in the program.

Our work differs in the level of granularity and on the aspects considered. Indeed, Lanza *et al.* [62] considered only file implementation to identify stability in different versions without considering information coming from version-control systems. Therefore, idle classes, for example, are those that did not change too much after their introduction in terms of source code and not in terms of commits. Thus, if a file changes frequently but without majors modifications in it implementation during the observation period, it will be identified as an "idle file", which is contradictory to its category name (idle files). We propose to identify file stability by mining program history. In the context of change analysis, if a file changes frequently, it will be identified as a "changed file".

2.3 Co-change: Definition and Detection

Ying *et al.* [111] and Zimmermann *et al.* [115] applied association rules to identify co-changing files. Their hypothesis is that past co-changed files can be used to recommend source code files potentially relevant to a change request. An association-rule algorithm extracts frequently co-changing files of a transaction into sets that are regarded as change patterns to guide future changes. Such an algorithm uses co-change history in CVS and avoids the source code dependency parsing process.

Ceccarelli *et al.* [20] and Canfora *et al.* [19] proposed the use of a vector auto-regression model, a generalisation of univariate auto-regression models, to capture the evolution and the inter-dependencies between multiple time series representing changes to files. They used the bivariate Granger causality test to identify if the changes to some files are useful for forecasting changes to other files. They concluded that the Granger test is a viable approach to change-impact analysis and that it complements existing approaches like association rules.

Antoniol *et al.* [5] presented an approach to detect similarities in the evolution of files starting from past maintenance. They applied the LPC/Cepstrum technique, which models a time evolving signal as an ordered set of coefficients representing

the signal spectral envelope, to identify in version-control systems the files that evolved in the same or similar ways. Their approach used cepstral distance to assess series similarity (if two cepstra series are "close", the original signals have a similar evolution in time).

Bouktif *et al.* [16] defined the general concept of change patterns and described one such pattern, Synchrony, that highlights co-changing groups of artefacts. Their approach used a sliding window algorithm as in [115] to build Synchrony change pattern occurrences. They also used the technique of Dynamic Time Warping developed in pattern recognition and adopted in speech recognition systems, to identify groups of co-changing files. Yet, the authors reported that their approach has an average precision and recall that could be significantly improved by using clustering techniques in addition to Dynamic Time Warping.

Discussion

Approaches based on association rules [111], [115], compute only the frequency of co-changed files in the past and they omit many other cases, *e.g.* files that co-change with some shifts among change periods. In Section 3.4, we showed that approaches based on association rules cannot detect all occurrences of co-change because, by their very definition, they do not integrate the analysis of files that are maintained by different developers and–or with some shift in time, which could lead to missed co-changing files.

Indeed, previous approaches could find files having very similar maintenance evolution history, but they did not present a tool to detect co-changed files maintained by different developers in periods of time of more than a few minutes. We were inspired by [16] to name the Asynchrony change pattern presented in this thesis. We also introduced a novel change pattern inspired from co-changes, the Dephase change pattern, that describes co-change among files with the some shifts in time.

Change propagation analyses how changes made to one file propagate to others. Law and Rothermel [64] presented an approach for change propagation analysis based on whole-path profiling. Path profiling is a technique to capture and represent a program dynamic control flow. Unlike other path-profiling techniques, which record intra-procedural or acyclic paths, whole-path profiling produces a

single, compact description of a program's control flow, including loops iteration and inter-procedural paths. Law *et al.*'s approach builds a representation of a program's behavior and estimates change propagation using three dependency-based change-propagation analysis techniques: call graph-based analysis, static program slicing, and dynamic program slicing.

Zhang *et al.* [112] conducted a case study in an industry project to investigate whether the dependency between individual requirements are useful in practise, in particular for change propagation analysis. They found that change propagation analysis is affected by a practitioner's viewpoint and experiences. The participant with project management experiences may emphasize the changes that happen at the business and high-level design level. The participants with requirements engineering experiences pay special attention on the changes at the level of requirements description and business. The participant with development experiences care more about the changes happening at the implementation level.

Hassan and Holt [42] investigated several heuristics to predict change propagation among source code files. They defined change propagation as the changes that a file must undergo to ensure the consistency of the program when another file changed. They proposed a model of change propagation and several heuristics to generate the set of files that must change in response to a changed file. Their approach uses: (1) frequency techniques to return the most frequently related files; (2) recency techniques to return files that were related in the recent past; and, (3) hybrid (frequency + recency) techniques.

Zhou *et al.* [113] 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 a program, such as change comments and author information. They used the Evolizer system that retrieves all modification reports from a CVS and uses a sliding window algorithm to group them.

Canfora and Cerulo [18] proposed an approach to derive the set of files impacted by a proposed change request. A user submits a new change request to a Bugzilla database. The new change request is then assigned to a developer for resolution, who must understand the request and determine the files of the source code that will be impacted by the requested change. Their approach exploits information

retrieval algorithms to link the change request descriptions and the set of historical source file revisions impacted by similar past change requests.

D'Ambros *et al.* [22] presented the Evolution Radar, an approach to integrate and visualise module-level and file-level logical couplings, which is useful to answer questions about the evolution of a program, the impact of changes at different levels of abstraction, and the need for restructuring. German [33] used the information in the CVS to visualize what files are changed at the same time and who are the people who tend to modify certain files. He presented SoftChange, a tool that uses a heuristic based on a sliding window algorithm to rebuild the Modification Record (MRs) based on file revisions. In Softchange, a file revision is included in a given MR if all the file revisions in the MR and the candidate file revision were created by the same author and have the same log. Beyer and Hassan [12] introduced the evolution story-board, a new concept for animated visualisations of historical information about the program structure, and the story-board panel, which highlights structural differences between two versions of a program. They also formulated guidelines for the usage of their visualisation by non-experts and to make their evaluations repeatable on other programs.

Discussion

We share with all the above authors the idea that change propagation identification into existing source code is a powerful mechanism to assess code maintainability. Their change-propagation models can be used to predict future change couplings, but they do not allow differentiation between different change patterns. All of these approaches grouped change couplings created by the same author and have the same log message; consequently, they cannot detect typical situation of co-changed file such as file maintained by different developers.

In addition, Xing and Stroulia [108], reported that visualisations approaches are limited in their applicability, because they assume a substantial interpretation effort by their users and they do not scale well: they become unreadable for large programs with numerous components.

2.4 Faults-proneness: Definition and Detection

Shah and Morisio [89] analysed the relationship of different complexity metrics with the faults for three categories of software projects i.e. large, medium and small. They observed that for some complexity metrics high complexity results in higher defects. They called these metrics as effective indicators of defects. For example, they reported that in the small category of projects, they found LCOM metrics, *i.e.* Lack of Cohesion of Methods which measures the correlation between the methods and the local instance variables of a class, as effective indicator in the medium category of project. They reported that complexity metrics relation to defects also varies with the size of projects.

Nagappan and Ball [74] performed a study on the influence of code churn [41], *i.e.* which measures the changes made to a component over a period of time, quantifies the extent of this change, on the fault density. They found that relative code churn was a better predictor than absolute churn. Moser *et al.* [73] used metrics (*e.g.* code churn, past faults and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse. Hassan and Holt [43] proposed heuristics to analyse fault proneness and they found that recently modified and fixed classes were the most fault-prone. Ostrand *et al.* [77] predict faults on two industrial programs, using change and fault data. Bernstein *et al.* [11] used fault and change information in non-linear prediction models. Zimmermann and Nagappan [114] used dependencies between binaries in Windows server 2003 to predict faults. Marcus *et al.* [70] used a cohesion measurement based on LSI for fault prediction. Neuhaus *et al.* [75] used a variety of features of Mozilla, such as past faults, package imports, call structure, to determine fault vulnerabilities.

Discussion

Previous approach on fault-proneness did not link artefact dependencies and artefacts evolution behaviors to faults. In this thesis, we link between (1) specific change patterns and fault-proneness, (2) artefact evolution and co-evolution trends and fault-proneness and, (3) and anti-patterns dependencies and fault-proneness.

2.5 Anti-patterns: Definition and Detection

The first book on "anti-patterns" in object-oriented development was written in 1995 by Webster [102]. In this book, the author reported that an anti-pattern describes a frequently used solution to a problem that generates ineffective or decidedly negative consequences. Brown *et al.* [17] described 40 anti-patterns, which are often described in terms of lower-level code smells. These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic audience. They are the basis of all the approaches to detect anti-patterns.

The study presented in this thesis relies on the anti-patterns detection approach proposed in [96]. However several other approaches have been proposed. For example, Van Emden *et al.* [97] developed the JCosmo tool. This tool parses source code into an abstract model (similar to the Famix meta-model). It used primitive and rules to detect the presence of smells and anti-patterns. The JCosmo tool can visualize the code layout and anti-patterns locations. The goal of this tool is to help developers assess code quality and perform refactorings. The main difference compared with other detection tools is that JCosmo tries to visualize problems by visualizing the design. Marinescu *et al.* developed a set of detection strategies to detect anti-patterns based on metrics [82]. Settas *et al.* explored the ways in which an anti-pattern ontology can be enhanced using Bayesian networks [88]. Their approach allowed developers to quantify the existence of an anti-pattern using Bayesian networks, based on probabilistic knowledge contained in an ontology.

The Integrated Platform for Software Modeling and Analysis (iPlasma) described in [63] can be used for anti-patterns detection. This platform calculates metrics from C++ or Java source code and applies rules to detect anti-patterns. The rules combine the metrics and are used to find code fragments that exceed some thresholds.

There are few papers analyzing empirically the anti-patterns relationships:

Binkley *et al.* [13] defined the dependence anti-pattern as a dependence structure that may indicate potential problems for ongoing software maintenance and evolution. Dependence anti-patterns are not structures that must always be avoided. Rather, they denote warnings that should be investigated. Typically these problems will take the form of difficulties in comprehension, testing, reverse engineering,

re-use, and maintenance. The authors showed how these anti-patterns can be identified using techniques for dependence analysis and visualization.

Vokac [101] analyzed the corrective maintenance of a large commercial program, comparing the defect rates of classes participating in design motifs against those that did not. Their approach showed correlation between some design patterns and smells like LargeClass but do not report an exhaustive investigation of possible correlations between these patterns and anti-patterns. Pietrzak and Walter [80] defined and analysed the different relationships that exist among smells and provide tips how they could be exploited to alleviate detection of anti-patterns. They proposed six coarse relations that describe dependencies between smells: plain support, mutual support, rejection aggregate support, transitive support, and inclusion.

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.* [14] 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.

Wei and Shatnawi [69] investigated the relation between the probability of a class being faulty and some anti-patterns based on three versions of Eclipse, showed that classes with the anti-patterns God Class, Shotgun Surgery, and Long Method have a higher probability to be faulty than other classes. They concluded that there was a need for broader studies to validate their results.

Olbrich *et al.* [76] 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 anti-patterns.

Khomh *et al.* [54, 95] studied the impact of classes with design defects (code smells and anti-patterns) on change proneness and fault proneness. They showed that classes participating in design defects are more change- and fault-prone than classes not participating in design defects.

Discussion

Previous approaches advanced the state-of-the-art in the specification and analysing of design defects and their relation with fault-proneness. In this thesis, we focus on detecting anti-patterns relationships with others classes to report the impact of such relationships on the fault-proneness of programs.

Indeed, we share with all the above authors the idea that anti-patterns detection is a powerful mechanism to asses code quality, in particular indicating whether the existence of anti-patterns and the growth of their relationships makes the source code more difficult to maintain.

Rather than focusing on the relationships among code smells and anti-patterns, our study focuses on analysing anti-patterns dependencies and their impact on fault-proneness.

While it is hard to define what a “bad” dependence structure should look like, we believe that it is comparatively easy to identify dependence between anti-patens and others classes in the programs that denote potential problems.

CHAPTER 3

IMPROVEMENT OF CO-CHANGE ANALYSIS

3.1 Introduction

As discussed in Chapter 2, several of the previous approaches identify co-changes among files, *e.g.*, [111], and [115], which represent the dependencies between files that have been observed to frequently change together. In the same context, Change patterns [16] are described as motifs that highlight co-changing groups of files [16] and that report the (often implicit) dependencies or logical couplings among files that have been observed to frequently change together [31].

In [47], we showed that previous approaches could not detect change patterns between files in long time intervals and–or performed by different developers and with different log messages, so they miss these varieties of co-changed files. Yet, we also showed that such new change patterns provide interesting information to developers. For example, in the Bugzilla of ArgoUML, the bug ID 5378[1] states, in relation to `ArgoDiagram.java`, that an "ArgoDiagram should provide constructor arguments for the concrete classes to create", which relates to `ModeCreateAssociationClass.java`. The bug report confirms that these two files have a relationship, which is hidden because we could not detect dependencies between these two files by static analysis. However, no previous approach can detect that these files co-changed because they were maintained by the same developer `bobtarling`, but their changes were always separated by a few hours. Knowing the dependency among these files is useful to a new developer that must change `ArgoDiagram.java`: she must, also, assess `ModeCreateAssociationClass.java` for change. Indeed, Vanya *et al.* [100] found that, depending on the commit practices used, a suitable time intervals between check-in timestamps of files has to be determined and leveraged to reliably approximate change sets.

Let `F1` and `F2` be two files of the same program, the scenario illustrated with ArgoUML could happen when a developer is in charge of a subset of a large program

[1] `http://argouml.tigris.org/issues/show_bug.cgi?id=4604`

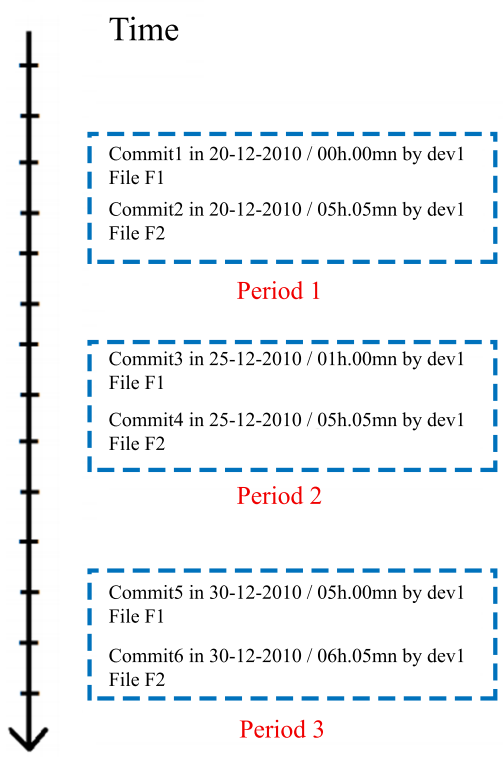

Figure 3.1 – `F1` and `F2` follow an Asynchrony change pattern.

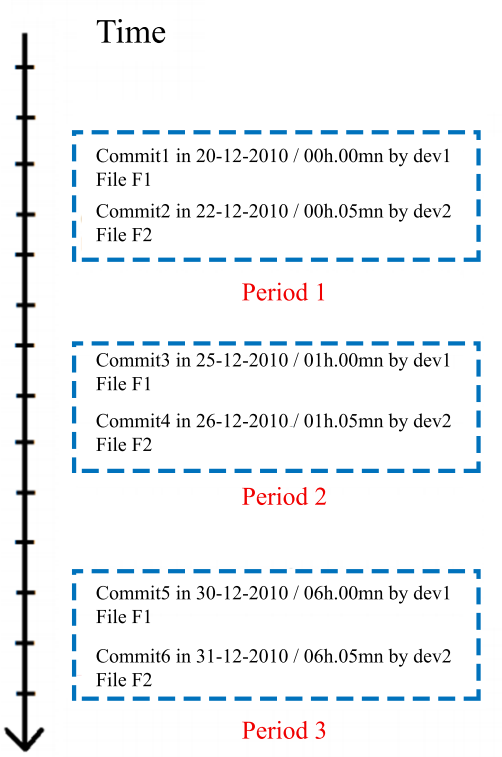

Figure 3.2 – `F1` and `F2` follow a Dephase change pattern.

composed of, among others, files `F1` and `F2`. She may change and commit these two files in the same day but with a few hours between each commit, as illustrated in Figure 3.1. This scenario may repeat for years and would be undetected by previous approaches, which use sliding windows of a few minutes to group changes committed by the same developer and with the same log message. Yet, such co-change situations contains important information both for the developer and her colleagues: to avoid introducing bugs into programs, changes to `F1` must likely propagate to `F2`, because these two files have a, possibly hidden, dependency.

As another example in ArgoUML, we found that the developers `mvw` and `tfmorris` contributed with some patches that contain `NotationUtilityJava.java` and `Model-`

`ElementNameNotationUml.java`[2] and the bug ID 2926[3] confirms that the two files have dependencies (see Section 3.4 for details). No previous approach can detect that these files co-changed because, during the development and the maintenance of ArgoUML, these two files were never changed by the same developer at the same time but were always changed by developers `mvw` and `tfmorris` in two consecutive times: first `NotationUtilityJava.java` and, subsequently, after some hours, `ModelElementNameNotationUml.java`, pointing out dependency among these files.

This previous scenario from ArgoUML happens when a developer D2 is reminded to change file `F2` to correct a bug after some time by developer D1, whenever D1 changed file `F1`, as illustrated in Figure 3.2. Previous work, *e.g.* [111], [115], [19], does not consider co-changed files if they were changed by two different developers in the same period even though knowing the, possibly hidden, dependency among such co-changed files could prevent developers from releasing a program with a bug because of a mismatch between files `F1` and `F2`.

In this thesis, we describe Macocha, an approach for detecting two novel change patterns [47] detailed in Section 3.2. Macocha, inspired from a previous work [16], defines and detects the Asynchrony change pattern, *macro co-changes* (MCC), and the Dephase change pattern, *dephase macro co-changes* (DMCC). It builds on previous work on co-changes and uses the concept of change periods, detailed in Section 3.2. It is defined as a set of changes committed by developers in a continuous period of time. In particular, we use the k-nearest neighbor algorithm [24] to group changes into their change periods.

The Asynchrony change pattern (*MCC*) describes a set of files that always change together in the same change periods. The Dephase change pattern (*DMCC*) describes a set of files that always change together with some shift in time in their periods of changes. We also consider approximate *MCC* and *DMCC* using the Hamming distance.

We formulate four research questions:

- *RQ1: How does Macocha compare to previous work in terms of number of changed files found?*

[2] `http://argouml.tigris.org/issues/showattachment.cgi/2118/20101116-patch-notation.txt`

[3] `http://argouml.tigris.org/issues/show_bug.cgi?id=2926`

- *RQ2: How does Macocha compare to previous work (association rules) in terms of precision and recall?*
- *RQ3: What is the precision and recall of Macocha when detecting occurrences of the Asynchrony change pattern?*
- *RQ4: How many occurrences of Dephase change patterns are in programs and how can they be useful for developers?*

We perform two types of empirical studies. *Quantitatively*, we compare the findings of Macocha with that of UMLDiff [106] and the co-change analysis of Macocha with the state-of-the-art association rules [111, 115] in term of precision and recall. *Qualitatively*, we use external information provided by bugs reports, mailing lists, and requirement descriptions to validate the Asynchrony and Dephase change patterns not found using previous approaches and to show that these novel change patterns explain real evolution phenomena and thus could help reduce maintenance costs. We apply our approach on seven programs: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, developed with three different programming languages, C, C++, and Java.

In this chapter, we describe our approach, Macocha, in more detail and we show how we use the k-nearest neighbor algorithm (KNN) to group changes into change periods and therefore to determine automatically the different duration of change periods. Then, we study the variations in precision and recall of our approach when using different values of its parameters and we perform a static analysis to validate the occurrences of new change patterns. Finally, we provide evidence on the relevance of the Dephase change patterns detected with different shifts in time.

This chapter is organised as follows: Section 3.2 presents Macocha. Section 3.3 describes our empirical study, while Sections 3.4 and 3.5 report and discuss its results as well as threats to validity. Section 3.6 concludes with future work.

3.2 Approach: Macocha

We now present the concepts of our approach using examples from ArgoUML. Macocha mines version-control systems (Concurrent Versions System named CVS[4] and Apache Subversion System named SVN[5]), to identify the change periods in a program, to group source files according to their stability through the change periods, and to identify the Asynchrony and the Dephase change patterns, *i.e.*, are macro co-changing or dephase macro co-changing.

3.2.1 Definitions

3.2.1.1 Change Period

A change period is a period of time during which several commits to different files occurred without "interruption", *i.e.*, these commits are separated by a few seconds or minutes. We conducted a case study to detect the duration of change periods for each subject. We need the concept of change period because the change periods (beginning dates and durations) differ across programs.

Consequently, we want to identify the change periods in a program by grouping all the changes committed closely together in time, independent of the developers who committed them and of their log messages. To identify changes occurring close to one another, *i.e.*, belonging to a same change period, we use the k-nearest neighbor algorithm (KNN). The KNN is a non-parametric learning algorithm[57] that does not make any assumptions on the underlying data distribution.

Hatton [44] presented an empirical study to estimate the time for the handling of a particular maintenance request (also known as change request) and showed that the largest duration of a change period to implement a maintenance request is not more than 40 hours. Therefore, we set the initial duration of the change period to 40 hours. However, as we noted earlier: these durations could change across subjects. Thus, we also report the variation of precision and recall using other values for the initial duration of the change period in Section 3.4. This variation

[4] `http://cvs.nongnu.org/`
[5] `http://subversion.apache.org/`

shows us, that for the seven subjects in this study, the 40 hours period yields the best results.

Our KNN-based algorithm to identify the change periods in a program history is illustrated in Figure 3.3 and consists of four separate phases:

- First, we divide the whole maintenance period, from first to last of the considered changes, of a given program, into equal sub-periods. Therefore, following Hatton, we divide the history of the program into change periods of equal duration of 40 hours.

- Second, to each change period, we assign the first change committed at a date nearest to, but no later than, the beginning date of the change period.

- Third, we use the KNN algorithm with $k = 2$ (we start the execution with $k = 2$ before testing other values of k) to assign the rest of the changes into their appropriate change periods: each change is assigned to the change period including its k nearest neighbors in terms of date of commit, whatever their developers and their log messages regardless of commit author and commit log message, and even if its change date is earlier than the beginning date of the change period.

- Fourth, when the KNN algorithm has assigned all changes into change periods, we recompute the beginning and end dates of each change period based on the dates of their earliest and latest changes. If there exists one or more change periods of duration greater than 40 hours, we reapply the KNN algorithm with an increased value of k, else we stop.

In Section 3.4, we show that using this algorithm to group changes into change periods allows us to improve precision and recall over the state-of-the-art association rules approach [115].

In ArgoUML. We find 290 change periods in two years of development. In Figure 3.4, we present the durations of all the different change periods detected in ArgoUML using the KNN algorithm. The mean duration of these change periods is around 27 hours 8 minutes, the standard deviation is around 12 hours 6 minutes. In addition, in Figure 3.4, we use box plots to display differences between the durations

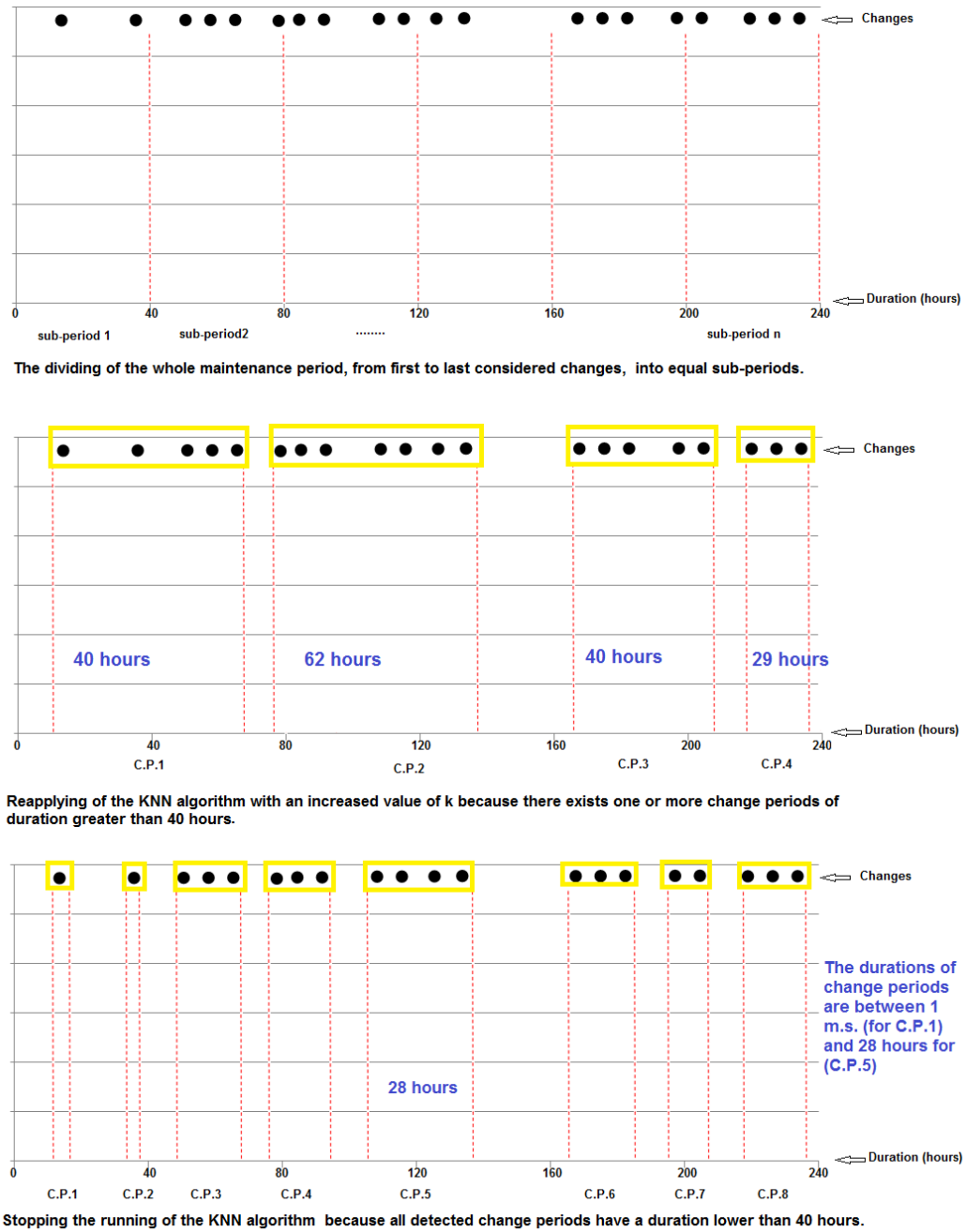

Figure 3.3 – Illustration of the KNN-based algorithm's steps to identify the change periods in a program history.

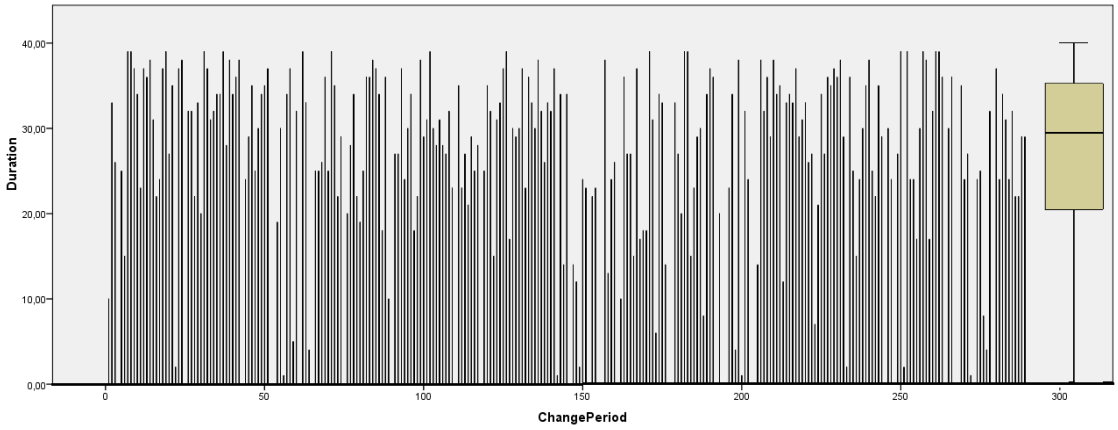

Figure 3.4 – The distribution of change period durations in ArgoUML detected by the KNN algorithm.

of change periods in ArgoUML without making any assumptions concerning the underlying statistical distribution. The spacings between the different parts of the box help indicate the degree of dispersion (spread) and skewness in the data, and identify outliers. Figure 3.4 is a box plot that summaries: the smallest observation (one millisecond), lower quartile (20 hours and 13 minutes), median (29 hours and 16 minutes), upper quartile (35 hours and 17 minutes), and largest observation (39 hours and 58 minutes).

3.2.1.2 Profile

We define a profile as a bit vector that describes whether a file is changed, or not, during each of the change periods of a program.

For each file in a program, its profile is defined as a vector $x = x_1...x_n$, where n represents the number of change periods determined by the KNN algorithm described above. The value of x_i indicates whether the file F is changed or not at the i^{th} change period:

$$x_i = \begin{cases} 1 & \text{if file is changed in the change period } i \\ 0 & \text{otherwise.} \end{cases}$$

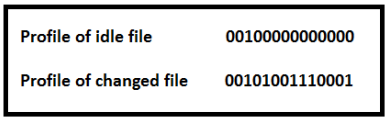

Figure 3.5 – Profiles for change periods of length $n = 10$ showing the changes committed in two different files.

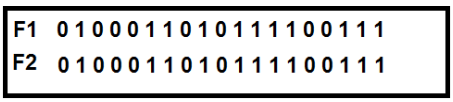

Figure 3.6 – Files `F1` and `F2` follow the Asynchrony change pattern.

We use bit vectors to model profiles because they allow efficient operations: setting a bit is constant in time, $\mathbf{O}(1)$; computing the union, the intersection of two bit vectors, or the complement of a bit vector, is linear in time, $\mathbf{O}(n)$.

3.2.1.3 File Stability

Macocha groups files according to their stability: idle and changed, as shown in Figure 3.5. Each group is a set of profiles with similar stability. *Idle files* do not change after their introduction into the program, while *changed files* are files that changed after their introduction into the program. Macocha uses this group to identify which *changed files* follow the Asynchrony or Dephase change patterns.

In ArgoUML. Macocha identifies 1,143 changed files from 1,621 files analyzed in two years of evolution of this program.

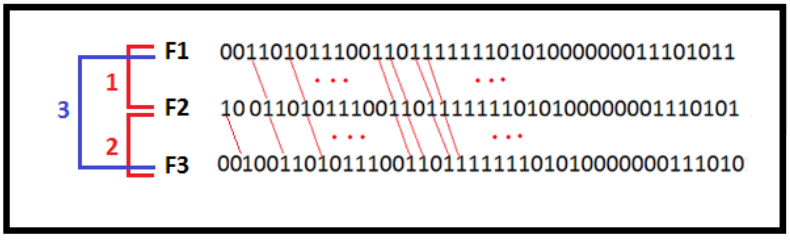

Figure 3.7 – Three different profiles showing an example of the Dephase change pattern.

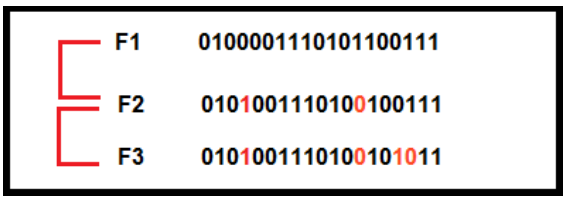

Figure 3.8 – Three different bit vectors showing approximate Asynchrony change pattern.

3.2.1.4 Change Patterns

Similar profiles grouped together represent occurrences of the Asynchrony and Dephase change patterns. Idles files do not change in any change period after their introduction into the program. Therefore, we do not consider this group of files because they are not useful for the co-change analysis due to their non-evolution.

Macocha returns the following sets of occurrences of change patterns:

- S_{MCC}, the set of macro co-changing files with identical profiles in a program;
- S_{DMCC}, the set of dephase macro co-changing files identified when shifting profiles by s change periods with $s \in [0, 5]$;
- S_{MCCH}, the set of approximate macro co-changing files with similar profiles in a program by using the Hamming distance with $D_H \in]0, 3[$;
- S_{DMCCH}, the set of approximate dephase macro co-changing files identified when shifting profiles by s change periods and by using the Hamming distance with $D_H \in]0, 3[$.

A S_{MCC} is two or more changed files that exactly change together with long time intervals between their changes and–or performed by different developers and with different log messages, *i.e.*, that have identical profiles during the life of a program, as illustrated in Figure 3.6. Given a file `F1`, a S_{DMCC} is the set composed of `F1` and one or more files, `F2...FM`, such that `F2...FM` always macro co-change with the same shift in time $s \in [0, n-1]$ with respect to `F1` during the evolution of a program. Thus, a Synchrony change pattern is a Dephase change pattern where $s = 0$.

Figure 3.7 illustrates that `F1` and `F2` are in dephase macro co-change with $s = 1$; `F2` and `F3` are in a DMCC with $s = 2$; and, `F1` and `F3` are in a DMCC with $s = 3$.

In ArgoUML. Macocha reports that `ProgressEvent.java` and `TestAction-AddEnumerationLiteral.java` followed a Dephase change pattern with $s = 2$; and, `ProgressEvent.java` and `IConfigurationFactory.java` followed a Dephase change pattern with $s = 3$. Previous approaches could not detect that these files follow such change patterns.

Macocha considers both identical and *similar* profiles (with or without shifts in time) to account for cases where the files did not change exactly in the same change periods. We use the Hamming distance [39] D_H to measure the amount of difference between two profiles, *i.e.*, the number of positions at which the corresponding bits are different. For a fixed length n, the Hamming distance is a metric on the vector space of the bit vectors of that length, as it fulfills the conditions of non-negativity and symmetry. The Hamming distance between two profiles a and b is equal to the number of ones in the the vector $a \oplus b$.

To determine the value of the appropriate Hamming distance for the detection of change pattern occurrences, we conduct a comparative study based on the evolution of the precision and recall when we use different values of D_H. In information retrieval contexts, precision and recall are defined in terms of a set of retrieved documents (e.g. the list of co-changed files produced by a query) and a set of relevant documents. Precision is the fraction of co-changed files that are relevant. Recall is the fraction of the co-changed files that are relevant to the query that are successfully retrieved. We also use the F-measure [85] to test accuracy. This measure considers both the precision and the recall of the test to compute the score. The traditional F-measure or balanced F-score (F1 score) is the harmonic mean of precision and recall:

$$F - measure = 2.\frac{Precision \times Recall}{Precision + Recall}$$

The F-measure score can be interpreted as a weighted average of the precision and recall, where an F-measure score reaches its best value at 1 and worst score at 0. We use the F-measure score when we compare the results of Macocha approach and the co-change detection approach based on association rules. We provide more

The Hamming distance	$D_H = 1$	$D_H = 2$	$D_H = 3$	$D_H = 4$	$D_H = 5$	$D_H = 6$
ArgoUML	44	353	439	499	528	621
FreeBSD	19	98	119	140	239	326
JFreeChart	116	425	502	533	647	801
Openser	7	41	50	82	128	166
SIP	102	570	598	652	703	788
XalanC	8	69	79	133	142	186
XercesC	11	125	158	196	206	290

Table 3.1 – Number of approximate Asynchrony change patterns detected using different values of the Hamming distance.

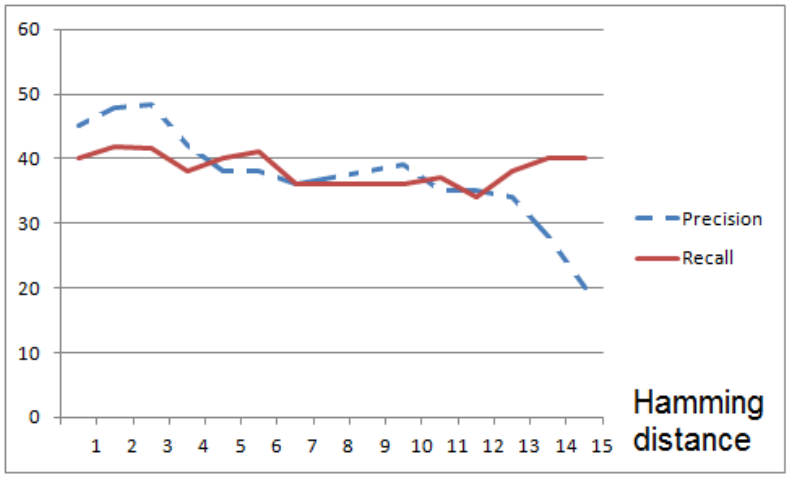

Figure 3.9 – The mean of precision and recall achieved by Macocha with different values of D_H for the seven programs.

details about the evaluation of our method in Section 3.4. After analysing several values of D_H between two profiles in different programs, we found that $D_H = 2$ is the best trade-off between precision and recall (as shown in Figure 3.9). Based on this finding, we will consider that two profiles are similar if the Hamming distance between them is equal to 1 or 2.

Table 3.1 reports the number of occurrences of approximate Asynchrony change pattern detected by Macocha with some values of Hamming distance D_H. For example, we find that with $D_H = 4$, Macocha reports 499 occurrences of approximate Asynchrony change pattern in ArgoUML. However, the precision and recall of Macocha with this value of D_H were less than 40%.

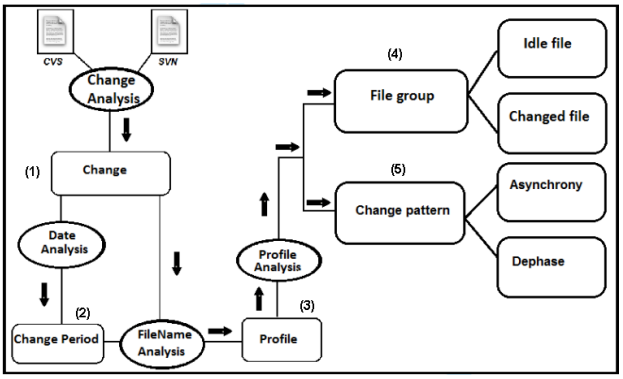

Figure 3.10 – Analysis-process.

Figure 3.8 illustrates that `F1` and `F2` are in approximate macro co-change with $D_H = 2$; `F2` and `F3` are in approximate *MCC* with $D_H = 2$; and, `F1` and `F3` are in a approximate *MCC* with $D_H = 4$.

In ArgoUML. Macocha reports that `ProgressEvent.java` and `ProgressListener.java` follow the Asynchrony change pattern (they have exactly the same profile), and `ProgressEvent.java` and `HelpListener.java` are in approximate *MCC* with $D_H < 2$.

3.2.2 Data Model, Implementation, and Outputs

A change contains several attributes: the changed file names, the dates of changes, the developers having committed the changes. Using this data, Figure 3.10 illustrates the concrete process of Macocha. This process takes as input a CVS or SVN change log. First, Macocha calculates the duration of different change periods using the *KNN* algorithm (1). Second, it groups changes in change periods (2). Third, it creates a profile that describes the evolution of each file in each change period (3). Fourth, it uses these profiles to compute the stability of the files (4) and, then, to identify changed files. Finally, Macocha detects and outputs macro co-changing files, i.e. changed files that follow the Asynchrony change pattern, and dephase macro co-changing files, i.e. changed files that follow the Dephase change pattern (5).

Macocha also weighs changes according to their distance in time because files co-changing frequently in the past, but not in recent times, may be less interesting than files having recently changed together. To do so, Macocha converts the bit vectors of files following an occurrence of a change pattern to a decimal number. Then, Macocha compares these decimals numbers to report the set of occurrences of change patterns including the most recently changed files (for files changed more recently, the conversion will naturally lead to greater decimal numbers). Ranking occurrences of the Asynchrony and Dephase change patterns allows us to detect the most recent occurrence of change patterns maintained in programs without having any impact on the precision and recall of results. Indeed, for practitioners, the consideration regarding the recency order is weighting changes according to their distance in time: files co-changing frequently in the past but not in recent times are not as interesting as files recently changing together, thus the time should be weighted. Previous work [22] already pointed this problem out and proposed techniques to deal with it. Thus, in this thesis, the analysis of change pattern recency does not have any impact on the results in terms of precision, recall and F-measure. The concrete examples reported in this empirical study include the most recently changed files.

3.3 Empirical Study

We use the Goal-Question-Metric (GQM) Approach [8] to define our empirical study. This approach is an approach on software metrics that defines a measurement model on three levels: conceptual level (goal), operational level (question) and quantitative level (metric). The approach is based upon the assumption that for an organization to measure in a purposeful way one must first specify the goals for itself and its projects, then one must trace those goals to the data that are intended to define those goals operationally, and, finally, one must provide a framework for interpreting the data with respect to the stated goals.

Following this approach, the goal of our study is to show that Macocha can identify occurrences of the Asynchrony and Dephase change patterns and that these occurrences describe file evolution and change propagation. Our purpose is to bring generalisable, quantitative evidence on the existence of these change

	ArgoUML	FreeBSD	JFreeChart	Openser	Sip	XalanC	XercesC
Languages	Java	C	Java	C	Java	C++	C++
# of Versions	9	11	5	5	16	13	14
# of Files	1,621	500	1,106	383	1,693	390	396
# of Commits	6,943	50,145	1,752	5,960	6,100	3,621	3,971
# of Developers	11	114	4	35	16	11	26

Table 3.2 – Descriptive statistics of the object programs.

patterns. The perspective is that both researchers and practitioners should be aware of the hidden dependencies among files to make informed changes. The context of our study is the maintenance of programs.

3.3.1 Research Questions

We formulate four research questions:

- *RQ1: How does Macocha compare to previous work in terms of number of changed files found?*
- *RQ2: How does Macocha compare to previous work (association rules) in terms of precision and recall?*
- *RQ3: What is the precision and recall of Macocha when detecting occurrences of the Asynchrony change pattern?*
- *RQ4: How many occurrences of Dephase change patterns are in programs and how can they be useful for developers?*

3.3.2 Objects

We choose seven programs developed with three different programming languages: ArgoUML, JFreeChart and Sip developed with java; FreeBSD and Openser developed with C; XalanC and XercesC developed with C++. Table 3.2 summarises program statistics. We use these programs because they are open source, have been used in previous work [19] [116], are of different domains and in different programming languages, span several years and versions, and underwent between thousands and tens of thousands of changes.

ArgoUML[6] is an UML diagramming program written in Java and released under the open-source BSD License. We analyse the evolution of this program for a period of two years, from 2007-02-19 to 2009-02-19. In this period, ArgoUML has gone through nine major versions. 11 developers participated in the maintenance of this program by committing 6,943 commits.

FreeBSD[7] is a free Unix operating system written in C and released under the open-source BSD License. We analyse the evolution of this program for a period of two years, from 2007-11-08 to 2009-11-08. In this period, FreeBSD has gone through 11 major versions. We notice that FreeBSD is the program that with the largest number of developers (114) and the largest number of commits (50,145) in this study.

JFreeChart[8] is a Java open-source framework to create complex charts in a simple way. We analyse the evolution of this program from 2008-02-13 to 2010-02-09. In this period, JFreeChart has gone through five versions. and 4 developers maintained its 1,106 files by committing 1,752 commits.

Openser[9] is an open source implementation of a SIP server, licensed under the GNU General Public License. This program can be used as a SIP registrar server, SIP router, SIP redirect server, etc. In addition, it can be used in small programs, for example in embedded programs like DSL routers, but also for large installations at Internet service providers with several million customers. The Openser project was created on 14 June 2005. We detect Asynchrony and Dephase change patterns in this program from this date to March 2007. In this period, Openser has gone through five versions and 35 developers maintained its 383 files by committing 5,960 commits.

SIP Communicator[10] is an audio/video Internet phone and instant messenger that supports some of the most popular VoIP and instant messaging protocols, such as SIP, Jabber, AIM/ICQ, MSN. SIP is open source and freely available under the GNU Lesser General Public License. It is written in Java. We analyse the evolution

[6] `http://argouml.tigris.org/`
[7] `http://www.freebsd.org/`
[8] `http://www.jfree.org/`
[9] `http://www.opensips.org/`
[10] `http://www.sip-communicator.org/`

of this program among 16 versions, from 2006-12-11 to 2008-12-08. 16 developers worked to maintained its 1,693 files by committing 6100 commits.

XalanC[11] is an open-source software library from the Apache Software Foundation written in C++. We analyse the evolution of this library for a period of two years, from 1999-02-21 to 2001-12-20. In this period, XalanC has gone through over 13 major versions and 11 developers maintained its 390 files by committing 3,621 commits.

XercesC[12] is a collection of software libraries written in C++ for parsing, validating, serialising, and manipulating XML. We analyse the evolution of this program from its publishing in 99-11-09 to 2001-11-09. In this period, XercesC has gone through 14 versions and 26 developers maintained its 396 files by committing 3,971 commits.

3.3.3 Analyses

To answer our research questions, we apply Macocha to the different subject programs and we collect all the occurrences of the Asynchrony and Dephase change patterns. We then perform two types of empirical studies. Quantitatively, we first compare in **RQ1** the results of Macocha with those of UMLDiff for file stability and we show that Macocha can identify the same idle and changed files as UMLDiff using only data from change logs. It does not produce as detailed information as UMLDiff but we can perform co-change analysis with data deducted from change logs. Second, we compare in **RQ2** the results of Macocha for detecting co-changing files with those of a state-of-the-art approach [115], that uses Association Rules. We also show that the set of macro co-changing files produced by Macocha includes the same co-changing files reported by Association Rules plus new co-changing files.

Qualitatively, we confirm in **RQ3**, that each of Macro co-change found by Macocha but not by the Association Rules-based approach [115] is indeed a dependency link, using external information from static analysis, bug reports, requirement descriptions, and mailing lists.

[11] `http://xml.apache.org/xalan-c/`
[12] `http://xerces.apache.org/xerces-c/`

We also validate in **RQ3** each occurrence Macro co-change patterns found by Macocha in the analysed programs by studying their static relationships (such as use relations, inheritance relations, and so on). For programs written in Java (ArgoUML, JFreeChart, and SIP), we use an existing tool, PADL [35], to automatically reverse-engineer class diagrams from the source code of object-oriented programs. A model of a program is a graph whose nodes are classes and edges are relationships among classes, such as: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [34]). As of today, PADL can only handle all static relations for programs written in Java. For programs considered in this study and written with other languages (FreeBSD, Openser, XalanC, and XercesC), we investigate static relationships among files following the Asynchrony and Dephase change patterns by manual verification.

Indeed, for each (approximate) occurrence of the change pattern, we confirmed in **RQ3** and **RQ4** the dependencies among their files by detecting their static relationships. If we could not detect such static relationships, we checked for other external information from bug reports and mailing lists. For static analysis, we use automatic tool detection of static relationships among files (Padl). For bug reports and mailing lists, we verify if these files are involving on bugs and mailing list between developers and we discuss the consistency of the external information before confirming the dependencies between these files.

We report a quantitative analysis in accordance with the state of the art, *i.e.*, we compare the findings of Macocha with that of UMLDiff [106] and the co-change analysis of Macocha with the state-of-the-art Association Rules [111, 115] in terms of precision and recall. We also report a qualitative analysis in accordance with external information and a static analysis, *i.e.*, we use external information provided by bugs reports, mailing lists, and requirement descriptions to validate the (approximate) Asynchrony and Dephase change patterns not found using previous approaches.

3.4 Study Results

We now present the results of our empirical study. Table 3.3 summarises the cardinalities of the sets obtained by applying Macocha.

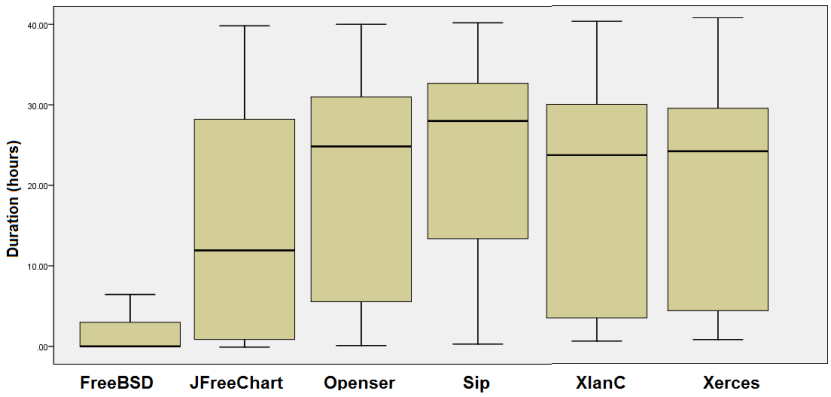

Figure 3.11 – Change period durations in different programs detected by the KNN algorithm.

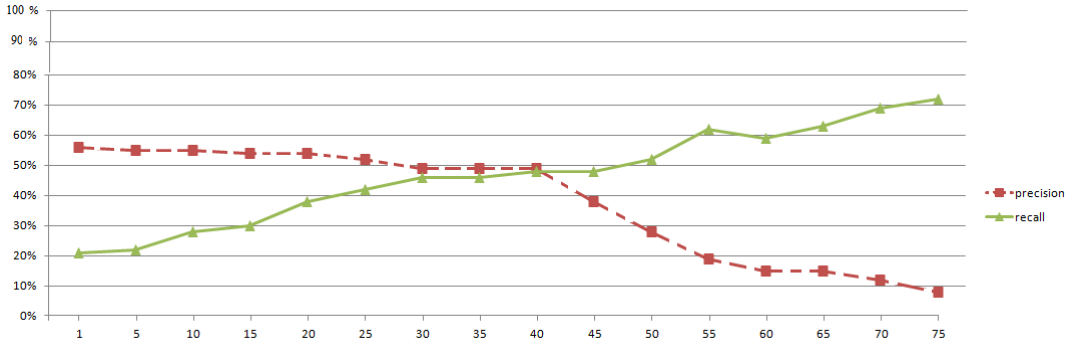

Figure 3.12 – The variation of precision and recall using other values for the initial duration of the change period.

3.4.1 *RQ1: How does Macocha compare to previous work in terms of number of changed files found?*

Macocha groups different commits in programs into change periods detected by the KNN algorithm. First, we observe that for the same duration of maintenance (two years for each program), the number of change periods detected by the KNN algorithm varies between 85 change periods, detected in FreeBSD, and 290 change periods in ArgoUML. Then, we use box plots (illustrated in Figure 3.11) to display differences between the durations of change periods in the seven analysed programs. We observe that, in each program, change periods detected by the KNN algorithm

	ArgoUML	FreeBSD	JFreeChart	Openser	SIP	XalanC	XercesC
Idle files	478	302	398	71	314	66	40
Changed files	1,143	198	708	312	1,379	324	356
# of S_{MCC}	192	45	281	21	350	41	68
# of S_{MCCH}	353	98	425	41	570	69	125

Table 3.3 – Cardinalities of the sets of Idle files, Changed files and (approximate) Asynchrony change pattern occurrences obtained in the empirical study.

do not have the same duration. For example, the duration of the change periods detected in JFreeChart varies between 1 millisecond (it is a single commit not clustered with any other commits) and 39 hours 57 minutes. In contrast, Figure 3.11 shows the distribution of different change periods detected in seven programs. Approximately 75% of the duration of change periods are shorter than 30 hours, so maintenance activities are dominated by small changes measured either by number or total time. We perform an analysis with varying values of the initial duration of change period using by the KNN algorithm to verify if we can prove empirically that, for t=40, we get the better precision/recall, and thus, confirm Hatton [44] observation. The variation of precision and recall using other values of the initial duration of the change period is illustrated in Figure 3.12. A higher value of the initial duration would yield a higher recall but a lower precision. While a smaller value of the initial duration would yield a higher precision but a lower recall.

By analysing each file that was changed or not during different change periods in programs, Macocha creates the set of profiles describing the evolution of different files in the whole life of the program. This analysis involves also eliminating idle files because they do not change in any change period after their introduction into the program. Therefore, they cannot participate in change patterns.

We distinguish idle from changed files by grouping together the files identified as short-lived and active by UMLDiff. Then, we compare the sets provided by UMLDiff and by Macocha and find that they are identical. For example, as shown in Table 3.4, Macocha finds 1,143 changed files in ArgoUML, identical to UMLDiff $414 + 729 = 1,143$ short-lived and active files. We note that the finer classification in three clusters of ArgoUML does not have any impact on the co-change analysis.

		Idle (Macocha)	Changed (Macocha)
ArgoUML	Idle Clusters (UMLDiff)	478	0
	Short-lived Clusters (UMLDiff)	0	414
	Active Clusters (UMLDiff)	0	729
JFreeChart	Idle Clusters (UMLDiff)	398	0
	Short-lived Clusters (UMLDiff)	0	43
	Active Clusters (UMLDiff)	0	665
SIP	Idle Clusters (UMLDiff)	314	0
	Short-lived Clusters (UMLDiff)	0	742
	Active Clusters (UMLDiff)	0	637
XalanC	Idle Clusters (UMLDiff)	66	0
	Short-lived Clusters (UMLDiff)	0	122
	Active Clusters (UMLDiff)	0	202
XercesC	Idle Clusters (UMLDiff)	40	0
	Short-lived Clusters (UMLDiff)	0	170
	Active Clusters (UMLDiff)	0	186

Table 3.4 – Cardinality of Macocha sets (idle groups and changed groups) in accordance with UMLDiff clusters [106].

Table 3.4 reports the number of idle, short-lived, and active files found by UMLDiff in the object-oriented subject programs (ArgoUML, JFreeChart, SIP, XalanC and XercesC) and their categorisation by Macocha. The main limitation for using UMLDiff is that it cannot detect idle, short-lived, and active files in programs developed with non object-oriented programming languages (FreeBSD and Openser developed in C), because it cannot create their UML-like representations. Machoca improves on UMLDiff in this respect and is able to analyse file stability for any program, providing that their CVS/SVN repositories are available.

Finally, Macocha computes file stability in a few minutes (unlike UMLDiff, which takes a few hours [107]), because it must create UML-like representations of the programs before performing evolution analysis. In the following, we present some examples from the object programs.

In JFreeChart For two years of maintenance, as shown in Table 3.5, Macocha found 131 change periods. In these periods, we detect 398 idle files. For example, the file `ColumnArrangement.java` was modified in only one change period. Using UMLDiff, we confirmed that this file belongs to an idle cluster.

We also detected 708 changed files. For example, the file `BarRenderer.java` was modified 17 times during the evolution of JFreeChart. Thus, this file belongs

to the changed group. Using UMLDiff, we confirmed that this file belongs to an active cluster.

In FreeBSD We found 302 idle files. For example, `kvmproc.c` was modified in one change period in two years.

We also detected 198 changed files. The file `ufsvnops.c` was modified in 15 change periods during the evolution of FreeBSD. We cannot use UMLDiff to verify this result, because it cannot analyse file stability in programs in C.

We answer *RQ1: How does Macocha compare to previous work in terms of number of changed files found?* as follows: Macocha detects the same number of changed files as UMLDiff, in the seven analysed systems, based on a CVS/SVN change log.

3.4.2 *RQ2: How does Macocha compare to previous work (association rules) in terms of precision and recall?*

For each program, Macocha detects files that have identical or similar profiles (the Macro co-change sets) and reports them in Table 3.3. For example, in ArgoUML, we detect 192 Asynchrony change pattern.

We compare the change patterns found by Macocha with the co-changing files found by an approach based on Association Rules [115] (see also [19]), which uses the Apriori algorithm [2] to compute Association Rules. The Apriori algorithm takes a minimum support and a minimum confidence and then computes the set of all Association Rules. To obtain a comprehensive set of rules, we consider as valid rules those achieving a minimum confidence of 0.9 as in previous work [115]. In this paper, Zimmermann *et al.* reported that with this value of confidence, their approach has the best precision and recall. We chose a minimum support of two to compare Association Rules and our approach (because in Macocha, changed files have at least two commits).

We denote the set of co-changing files found by an approach based on Association Rules [115] as S_{AR} and we compare the approximate macro co-changing files found by Macocha with the co-changing files found by an approach based on Association Rules to evaluate the performance of Macocha.

	Training Set				Testing Set			
	Start	End	# T	# CPs	Start	End	# T	# CPs
ArgoUML	07-02-19	09-02-19	4,718	290	09-02-22	11-02-21	2,225	191
FreeBSD	07-11-08	09-09-22	23,944	85	09-12-21	11-10-31	26,201	73
JFreeChart	08-02-13	10-02-09	1,555	131	10-02-16	12-02-13	197	24
Openser	05-06-14	07-06-04	2,321	247	07-06-05	09-06-15	3,639	281
SIP	06-12-11	08-12-08	2,870	261	08-12-09	10-12-09	3,230	307
XalanC	99-12-21	01-12-20	2,242	219	01-12-20	03-12-28	1,379	165
XercesC	99-11-09	01-11-09	1,820	204	01-11-12	03-11-08	2,151	227

Table 3.5 – Internal evaluation of Macocha (CPs: Change periods; T: Transactions).

	Macocha			Association Rules		
	Precision	Recall	F-measure	Precision	Recall	F-measure
ArgoUML	49%	30%	0.37	28%	29%	0.28
FreeBSD	19%	40%	0.45	11%	40%	0.17
JFreeChart	16%	33%	0.21	15%	33%	0.20
Openser	51%	32%	0.39	50%	32%	0,39
SIP	50%	55%	0.52	50%	52%	0.50
XalanC	82%	34%	0.48	79%	33%	0.46
XercesC	72%	56%	0.63	58%	46%	0.51
All programs	49%	48%	0.48	42%	38%	0.39

Table 3.6 – Internal evaluation of Macocha in comparison to an approach based on Association Rules [115].

In the context of the evaluation of co-change analysis methods, in an internal evaluation, we compare groups of co-changing files change-patterns extracted from a testing set of data that are not involved in the grouping process. An external evaluation is the evaluation of the accuracy of one approach by comparing their results with the result of other approaches or the observation of an expert. We perform an *internal evaluation* similar to that of Zimmermann *et al.*'s [115]. Given snapshots S_i, $i \in [1, ..., n]$, we build two sets $T_{train} = \{S_1 ... S_t\}$ and $T_{test} = \{S_{t+1} ... S_n\}$, as shown in Table 3.5. We use T_{train} to build Association Rules and macro co-change dependencies and we compare the co-changing files in T_{train} with those in T_{test}. Indeed, Macocha checks if files with similar (or same) profiles in T_{train} have similar (or same) profiles in the T_{test}.

For the seven programs, we observe that Macocha improves precision and recall over Zimmermann's approach based on Association Rules, as shown in Table 3.6.

For example, for ArgoUML, results indicate that the precision and the recall of Macocha, respectively 49% and 30%, are better than those of Association Rules, respectively 28% and 29%. For the sum of the objects considered in this study, the improvement in precision is larger than that for recall. Indeed, Macocha has +7% precision and +10% recall over Association Rules, *i.e.*, the precision and the recall of Macocha, respectively 49% and 48%, are better than those of Association Rules, respectively 42% and 38%. Thus, the F-measure value of Macocha, 0.48 is better than the F-measure value of Association Rules, 0.39.

We observed that the Apriori algorithm generates high support sets of rules that are later checked for high confidence. Therefore, high confidence rules with low support are not generated [2], which could lead to missed co-changing files.

We answer *RQ2: How does Macocha compare to previous work (association rules) in terms of precision and recall?* as follows: Macocha improves the identified co-changes over an approach based on Association Rules in terms of precision and recall, *i.e.* Macocha has +7% precision, and +10% recall over Association Rules.

3.4.3 *RQ3: What is the precision and recall of Macocha when detecting occurrences of the Asynchrony change pattern?*

The rationale for an internal evaluation is that no expert, no oracle and no pre-existing groups of co-changing files are available. Precision and recall are measured for the testing sets by considering, for each file, the groups resulting from the training sets as oracles. Such an internal validation has some limits [99] [22]: (1) Files co-changing frequently in the past (training set) but not recently (test set) will be considered wrongly as false negatives; (2) Files co-changing frequently recently (test set) but not in the past (training set) will be considered wrongly as false positives; (3) If the training set contains false positives or negatives, they cannot be detected using the testing set. In fact, that explain the somewhat low values of precision, recall and F-measure reported in Table 3.6.

To overcome the limits of an internal validation and to validate change patterns not found using Association Rules, we also performed an *external evaluation* of Macocha by considering the results of the Association Rules as an oracle and by manually validating the sets for differences between Macocha and Association

Rules. Because no expert and no pre-existing groups of co-changing files are available as an oracle, and because the high number of co-changed files detected in the seven subject programs, we chose, first of all, to compare Macocha findings and Association Rules findings. Second, we applied a static analysis to validate the MCCs not detected by the approach based on Association Rules because co-change analysis is known to be more useful when combined with static analysis [42]. For each (approximate) occurrence of the Asynchrony change pattern, we confirmed the dependencies among their files by detecting their static relationships. If we could not detect such static relationships, we checked for other external information from bug reports, mailing lists, and so on to validate the MCCs detected by Macocha.

For each set returned by the Association Rules-base approach, if an identical set was returned by Macocha, it was considered a true positive. If the two sets were not identical, we used external information to validate missing files and to decide if they presented a true positive, a false negative, or a false positive. For example, in JFreeChart, all the sets detected using Association Rules were detected by Macocha except nine sets. We detected static relationships among files of seven sets from the nine missed sets and we confirmed the dependencies among the files of the last two sets using bug reports in the Bugzilla of JFreeChart.

For Xerces, Table 3.6 describes the internal validation, it says that Macocha has +14% in precision, and +10% in recall over Association Rules. It means, for the testing set, Macocha has less false negatives and false positives than Association Rules. In Table 3.7, Maccocha has 100% precision and recall versus Association Rules. It means that Macocha and Association Rules detected exactly the same sets in the training set. Table 3.7 also reports, under the Association Rules header, the precision and recall of Macocha with respect to the approach based on Association Rules [115]. The precision and recall presented in Table 3.7 are measured relative to another method. So, precision really means "number of cases that Macocha said were true/false, that also Association Rules said were true/false", and recall really means "number of cases that Association Rules found, that also Macocha found". Table 3.7 shows that Macocha detects the majority of co-changing files detected using Association Rules in the seven object programs. In addition, Macocha detects other occurrences of change patterns not detected using Association Rules.

	Macocha vs. Association Rules	
	Precision (*)	Recall(**)
ArgoUML	94%	98%
FreeBSD	82%	100%
JFreeChart	65%	96%
Openser	95%	89%
SIP	98%	100%
XalanC	100%	99%
XercesC	100%	100%

Table 3.7 – Evaluation of Macocha when using the results of an approach based on Association Rules [115] as oracle (*: number of cases that Macocha said were true/false, that also Association Rules said were true/false; **: number of cases that Association Rules found, that also Macocha found).

Table 3.8 and Table 3.9 report, the adjusted precision and recall values of Macocha after manual validation, which show that Macocha detects occurrences of change patterns missed or wrongly reported using Association Rules. For example, in Openser, 89% of co-changing files found by Macocha were detected by Association Rules. While 95% of co-changing files detected by Association Rules were detected by Macocha. This comparison gives us 14 cases of false positives and 5 cases of false negatives, as shown in Table 3.8 and Table 3.9. We confirmed all of these cases by a static analysis of source code of these files performed manually. Indeed, a smaller value of change period duration would yield a higher recall but without detecting novel change patterns such as Asynchrony change patterns because we do not integrate the analysis of files that are maintained by different developers and–or with some delay in time, which could lead to missed co-changing files and change propagation scenarios.

In the following, we describe some occurrences of the Asynchrony change pattern that are missed by the previous approach and justify why they are missed and why it is important to detect them. We chose these examples from the most-recently changed files following the (approximate) Asynchrony change pattern.

In ArgoUML `SelectionActionState.java` and `SelectionState.java` followed the same occurrence of an Asynchrony change pattern. On the one hand, by using PADL [35] to automatically reverse-engineer class diagrams from the source

	External Validation of S_{MCCH} - S_{AR}				
	False positives	V.S.A	Bugs	Mails	Precision
ArgoUML	11	3	4	4	100%
FreeBSD	16	10	0	4	88%
JFreeChart	13	3	6	4	93%
Openser	14	5	0	0	100%
Sip	12	4	6	2	100%
XalanC	0	0	0	0	100%
XercesC	0	0	0	0	100%

Table 3.8 – Adjusted precision of Macocha when using the results of an approach based on Association Rules [115] as oracle and after manual validation using external information and static analysis (V.S.A: Validation by static analysis; S_{MCCH}: The set of approximate macro co-changing files with similar profiles in a program by using the Hamming distance; S_{AR}: The set of co-changed files found by an approach based on Association Rules).

	External Validation of S_{MCCH} - S_{AR}				
	False negatives	V.S.A	Bugs	Mails	Recall
ArgoUML	56	49	2	4	99%
FreeBSD	0	0	0	0	100%
JFreeChart	10	6	2	2	100%
Openser	5	5	0	0	100%
Sip	0	0	0	0	100%
XalanC	2	0	1	1	100%
XercesC	0	0	0	0	100%

Table 3.9 – Adjusted Recall of Macocha when using the results of an approach based on Association Rules [115] as oracle and after manual validation using external information and static analysis (V.S.A: Validation by static analysis S_{MCCH}: The set of approximate macro co-changing files with similar profiles in a program by using the Hamming distance; S_{AR}: The set of co-changed files found by an approach based on Association Rules).

code of ArgoUML, we detected a static dependency among these two files. On the other hand, in the Bugzilla of ArgoUML, the bug ID 2552[13] states that "we should use 3 state [...] this model could simply be the Action" in relation with these two files. By applying the co-change analysis for "Error Prevention" described in [115], we could not find co-change dependencies among them. Thus, we could not explain and–or predict bugs in relation to these two files.

JFreeChart `AbstractXYDataset.java` and `RenderAttributes.java` were in MCC. This is confirmed in the Bugzilla of JFreeChart by the bug ID 1654215[14] relating these two files. In fact, this bug reports that "Adding renderer with no dataset causes exception" and confirms static dependencies detected after analysing JFreeChart source code by PADL. These two files were changed by the same developer in a time-window of more than a few minutes. Therefore, by applying the Association Rules-based approach described in [115], we could not not find that these files were co-changing. Consequently, we could not give to the developer the knowledge about dependencies among these two files to maintain them properly, as described in [111, 115].

We answer *RQ3: What is the precision and recall of Macocha when detecting occurrences of the Asynchrony change pattern?* as follows: Macocha detects change patterns missed or wrongly reported using Association Rules. Macocha has a mean precision of 97% and a mean recall of 99% for Asynchrony change patterns detection.

3.4.4 *RQ4: How many occurrences of Dephase change patterns are in programs and how can they be useful for developers?*

No previous approach could detect files maintained with similar trends and some given shifts in time. Indeed, the Dephase change pattern is the main contribution of the thesis in term of novelty. We confirmed the existence of occurrences of the (approximate) Dephase change patterns by detecting static relationships among their files. We also confirmed these occurrences using external information. Table 3.10 illustrates the number of occurrences of the (approximate) Dephase change

[13] `http://argouml.tigris.org/issues/show_bug.cgi?id=2552`

[14] `http://sourceforge.net/tracker/index.php?func=detail&aid=1654215&group_id=15494&atid=115494`

pattern detected and confirmed using external information and static analysis, in each program. We recall that an Asynchrony change pattern is a Dephase change pattern with $s = 0$.

Macocha can detect occurrences of the Dephase change pattern with several values of shift s. After analysing different sets of Dephase macro co-changing files detected in the seven object programs, we observed that the number of occurrences of the Dephase change pattern detected by Macocha and confirmed by external information in the majority of subjects decreased from $s = 3$ and is close to 0 from $s = 5$, as shown in Table 3.10. In our case study, we detected Dephase macro co-changes for $s \in [1, 5]$ to obtain an accurate set of results. In Table 3.10 we note that, for example, out of the 27 occurrences of Dephase change patterns detected in ArgoUML with a shift s=2, 24 occurrences were confirmed by static analysis. In our case study, the precision was 88%. As other examples derived from Table 3.10, we note the small number of (approximate) Dephase change pattern occurrences detected in programs developed in C or C++ (less than six occurrences for each value of the shift s).

We now report some typical occurrences of the (approximate) Dephase change pattern from different programs with different values of shift s. We chose these examples from the most-recently changed files following the (approximate) Dephase change patterns.

In FreeBSD We find that `ip-fw2.c` and `sysv-msg.c` follow the same occurrence of a Dephase change pattern with a shift $s = 2$. After manually investigation of the source code of these two files we do not find any static relationships among them. However, in the mailing list of FreeBSD, the `Message-ID: <201-20107201823.H3704@sola.nimnet.asn.au>` states that the two files were used to implement the same requirement of "ruleset sequence" in a lengthy message from a developer about "IPFW transparent VS dummynet rules". Our case study confirms dependencies among these two files by external information.

In SIP We find that `Html2Text.java` and `FileTransfReceiveListener.java` were changed systematically with five shift periods in two years. Therefore, they followed the Dephase change pattern with a shift $s = 5$. These two files implement

	Shift	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 5$
ArgoUML	Approximate *DC*	46	27	39	51	20
	Approximate *DC* confirmed	32	24	30	39	2
	DC	3	5	5	3	1
	DC confirmed	3	5	4	2	0
FreeBSD	Approximate *DC*	1	3	1	5	6
	Approximate *DC* confirmed	1	3	1	4	2
	DC	0	1	0	1	0
	DC confirmed	0	1	0	1	0
JFreeChart	Approximate *DC*	46	14	9	10	9
	Approximate *DC* confirmed	43	14	8	7	5
	DC	6	1	3	0	0
	DC confirmed	5	1	1	0	0
Openser	Approximate *DC*	1	8	6	7	6
	Approximate *DC* confirmed	1	6	6	5	2
	DC	1	1	0	1	1
	DC confirmed	1	0	0	1	0
Sip	Approximate *DC*	43	68	89	83	96
	Approximate *DC* confirmed	43	66	80	62	44
	DC	6	4	4	6	9
	DC confirmed	5	3	4	5	7
XalanC	Approximate *DC*	1	0	2	0	0
	Approximate *DC* confirmed	1	0	0	0	0
	DC	0	1	0	1	0
	DC confirmed	0	0	0	1	0
XercesC	Approximate *DC*	1	1	1	0	0
	Approximate *DC* confirmed	1	1	1	0	0
	DC	1	0	1	0	0
	DC confirmed	1	0	0	0	0

Table 3.10 – Evolution of the number of occurrences of (Approximate) Dephase change patterns, *DC*, detected and manually confirmed by static analysis for different values of shift s.

the same feature[15]: "Instant Messaging". By performing a static analysis, we detected a static dependency among these two files. Therefore, we confirmed the occurrence of the Dephase change pattern formed by these two files.

In XercesC We found that `XercesXPath.cpp` and `XMLDateTime.cpp` follow the same occurrence of an approximate Dephase change pattern with shift $s = 1$. This change dependency is confirmed by multiple static relationships detected when we examine the source code of these two files. In addition, in the mailing list of XercesC, a message[16] on April 1, 2009 about "a legitimate bug with the time of day" states that these two files are related.

Indeed, our approach guides programmers based on the program history. Suppose the developer changed a file F1 in the program. Macocha then suggests to change the file F2 because in the past, both items always have been changed together with same shift in time. All Macocha needs is a CVS/SVN repository. The benefit of Macocha is that it points out item coupling that is undetectable by previous co-change analysis such as between files maintained by different developers or–and with some shift in time. In the following, we show how Dephase macro co-changing files detected using Macocha support the following three scenarios:

3.4.4.1 Scenario 1: Management of Development Teams

We think that if two files follow the same occurrence of a Dephase change pattern, they probably should be maintained by the same team of developers to minimise the risks of introducing bugs in the future. Otherwise, if it is obligatory to have different teams to maintain files that they follow the same occurrence of a Dephase change pattern, *i.g.* in the case of a feature that it is developed by a product team and its tests are developed by the test team, these teams should exchange information about these files after each change.

We notice that they might probably send each other private emails notifying the other party of a change. In our context, it is neither possible to conclude that developers are sending each other private emails notifying the other party, nor is the opposite true. We decide to perform, as future work, an empirical

[15] `http://www.jitsi.org/index.php/Main/Features`
[16] `http://markmail.org/message/a5secbiwkgxtegxb`

study to verify such fact by asking developers of these programs. The team of developers most likely possesses a wealth of unwritten knowledge about the design and implementation choices that they made for these files, which would help them to prevent introducing bugs [83].

Consequently, a team leader could redefine the organisation of the maintenance team according to the Dephase macro co-changes links among files, so that her team does not introduce bugs because of the absence of information or lack of communication among developers. For example, in ArgoUML, when we analysed changes made in three Dephase macro co-changing files that have generated bugs (BugID 1957 BugID 2926, and BugID 4604), we found that these changes have been made with one shift in time in their periods of change and by different developers. These co-changes cannot be detected by previous co-change analysis approaches. Thanks to Dephase macro co-change, a team leader could ensure that the team who will maintain these files in each change period has the necessary knowledge to maintain the dependency among these files.

3.4.4.2 Scenario 2: Bug and Change Propagation

If co-change dependencies are not properly maintained, developers could introduce bugs to a program [23]. Knowing that two files are in Dephase macro co-change implies the existence of (hidden) co-change dependencies between these two files. With our approach, for each program studied, we detected files in Dephase macro co-changes. By using external information, we confirmed our observation and that some of these files indeed participate to bugs. For example, in SIP, we detected seven bugs in relation with Dephase macro co-changing files. By applying the association rule approach described in [115], we could not find that these files are co-changing. A full list of defects belonged to Asynchrony and Dephase change patterns detected in the seven analysed programs is available on-line at http://www.ptidej.net/downloads/experiments/jsme12/. Therefore, by knowing files that are in *DMMC*s, Macocha provides the list of files that developers should be carefully considered by developers to ensure the change propagation and the proper maintainability.

3.4.4.3 Scenario 3: Traceability Analysis

The change history represents one of sources of information available for recovering traceability links that are manually created and maintained by developers [50]. The version history may reveal hidden links that relate files and would be sufficient to attract the developers' attention [51]. For example, in SIP, we detect traceability links between four approximate Dephase macro co-changing files. By applying the association rule approach described in [115], we could not find that these files are co-changing.

Due to the distributed collaborative nature of open-source development, version-control systems are the primary location of files and the primary means of coordination and archival [51]. The requirements of open-source programs are typically implied by communication among project participants and through test cases. However, such traces of requirements are lost in time. In a previous work [3], Ali *et al.* presented an approach, Histrace, that used CVS/SVN change logs to build traceability links between high-level documentation and source code entities, observing that log messages are tied to changed entities and, thus, can be used to infer traceability links. Histrace improved with statistical significance the precision of the traceability links, while also improving recall but without statistical significance. The authors thus showed that their trust-based approach indeed improves precision and recall and also that CVS/SVN change logs are useful in the traceability recovery process. Ongoing work includes using the (approximate) Dephase macro co-change to improve traceability links between files in the same system.

We answer *RQ4: How many occurrences of Dephase change patterns are in programs and how can they be useful for developers?* as follows: in all object programs, Macocha detects occurrences of the (approximate) Dephase change pattern, *e.g.*, 183 occurrences were detected in ArgoUML, while these occurrences specify change propagation as well as they can help to reorganize maintenance tasks by spotting hidden dependencies among files.

3.5 Discussions

3.5.1 Observations

In **RQ1**,we reported that Macocha can identify changed files before performing the co-change analysis. On the one hand, we showed that Macocha was able to analyse file stability for any program, providing that their CVS/SVN repositories are available. On the other hand, Macocha computed file stability in a few minutes (unlike UMLDiff, which takes few hours [107]). It could be true that a baseline approach (a file is idle if it never changes 40 hours after its introduction) could perform pretty well and report similar results without needing the additional complexity. However, with a baseline approach, we could not apply the other steps of our approach that allow for the detection of new change patterns: Asynchrony and Dephase change patterns. Indeed, we performed the first step to eliminate files that do not correspond to a meaningful modification task.

A major application for co-change detection approaches is to guide users through source code. The user changes some entity and these approaches recommend possible future changes in a view. To evaluate the predictive power of a co-change detection approach in this situation, the authors of a previous approach based on association rules [115] tested the capability of their approach to predict future changes. For each transaction T, and each entity **e** belong to entities(T), they queried Q = e, and checked whether the approach would predict E = entities(T) - e. For each transaction, they thus tested entities(T) queries, each with one element. We repeat the same analysis in **RQ2**. In fact, Zimmermann *et al.* [115] applied association rules to identify co-changing files and showed that increasing the support threshold also increases the precision, but decreases the recall as their approach gets more cautious. However, using the highest possible thresholds does not always yield the best precision and recall values. If they increased the confidence threshold above 0.80, both precision and recall decrease. Furthermore, Zimmermann *et al.* showed that a high support and confidence threshold is required for high precision. Still, such values result in a very low recall, indicating a trade-off between precision and recall. In our study conducted in this chapter, we showed that their approach can predict 38% of all entities changed later in the same transaction. While, Macocha can predict 48% of all entities changed later in the same transaction for the

same programs. Approaches based on association rules compute only the frequency of co-changed files on individual commits and omit many other cases, *e.g.* files that co-change with some shifts among change periods.

In **RQ3** and **RQ4**, we showed some gains of our approach compared to previous co-change analysis approaches. For example, approaches based on association rules cannot detect all occurrences of co-change and any occurrences of DMCCs because, by their very definition, they do not integrate the analysis of files that are maintained by different developers and–or with some shift in time, which could lead to missed co-changing files and change propagation scenarios.

The main contribution in this chapter is detecting several occurrences of the (approximate) Asynchrony and Dephase change patterns (two novel change patterns) in different programs belonging to different domains and with different sizes, histories, and programming languages. We do not detect MCCs and DMCCs with the same proportions in each program. We observe that the proportion of MCCs and DMCCs found in the programs developed in Java (ArgoUML, JFreeChart, and SIP) are greater than the proportion of MCCs and DMCCs found in programs developed in C or C++ (see Table 3.3 and Table 3.10). We explain this observation by the fact that, on the one hand, the majority of FreeBSD files are idle, FreeBSD is the largest system in term of the number of commits (50,145 commits) and the number of developers (114 developers), and that may have an impact on the non-organization of maintenance tasks. On the other hand, Openser, XalanC, XercesC are the smallest object programs (having less than 400 files). We also apply our approach to detect (Dephase) macro co-changes on fewer C and C++ files than Java files, which could explain the lower number of MCCs and DMCCs.

3.5.2 Threats to Validity

Some threats limit the validity of the results of our empirical study.

Construct Validity

Construct validity threats concern the relation between theory and observations. In this study, they could be due to implementation errors. They could also be due to a mistaken relation between changed files. We believe that this threat is mitigated by the facts that many authors discussed this relation, that this relation

seems rational, and that the results of our analysis shows that, indeed, MCCs and DMCCs exist and are corroborated by external sources of information (bug reports and others). In addition, our results can still be affected by the presence of false negatives, *i.e.*, by a low recall exhibited by the co-change detection approach. As previous work detected co-changes committed by the same author in a short time window, relaxing these constraints may also lead to false positives. The results of our empirical study show that Macocha improves precision and recall with respect to the state of the art in seven different programs. However, we cannot claim that our approach will give similar results for any program.

Internal Validity

Internal validity is the validity of causal inferences in studies based on experiments. The internal validity of our study is not threatened because we have not manipulated a variable (the independent variable) to see its effect on a second variable (the dependent variable).

Reliability Validity

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to re-implement our approach and replicate our empirical study. Moreover, both ArgoUML, FreeBSD, JFreeChart, Openser, Sip, XalanC, and XercesC source code repositories are publicly available. the way our analysis were performed is described in detail in Section 3.2. The change logs, the list of bugs and the changed files of the seven programs analysed with their profiles to obtain our observations are available on-line at `http://www.ptidej.net/downloads/experiments/jsme12/`.

External Validity

We performed our study on seven different real programs belonging to different domains and with different sizes, histories, and programming languages. Yet, we cannot assert that our results and observations are generalisable to any other programs, and the fact that all the analysed programs are open-source may reduce this generability. Nevertheless, it would be desirable to analyze further systems, also developed in different programming languages, to draw more general conclusions. Future work includes replicating our study in other contexts and with other programs.

3.6 Summary and Lessons Learned

The development and maintenance of a program involves handling large numbers of files. These files are logically related to each other and a change to one file may imply a large number of changes to various other files. Many previous works try to reduce program maintenance costs by detecting and using co-changing files. For example, the authors in [16] defined the Synchrony change pattern as common and recurring modifications of programs' files in time.

In this chapter, we introduced the Asynchrony change pattern and the Dephase change pattern, as well as their approximate versions, to explain other scenarios of co-change and change propagation, which could help developers to maintain a program's files appropriately. We proposed an approach, Macocha, which mines software repositories and uses several algorithms and techniques (k-nearest neighbor algorithm, the Hamming distance, and a bit vector model) to discover occurrences of the (approximate) Asynchrony and Dephase change patterns.

Macocha relates to file stability and co-changes. We therefore performed two types of empirical studies. Quantitatively, we compared Macocha with UMLDiff [106] and an association rules-based approach [115] by applying and comparing the results of the three approaches on seven programs: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC, and we showed that Macocha has better precision and recall than the state-of-the-art approaches based on association rules [111, 115]. Qualitatively, we used external information and static analysis to show that detected *MCC*s and *DMCC*s explain real, important evolution phenomena. We also showed that occurrences of Dephase change patterns do exist and help in explaining bugs, managing development teams, and performing traceability analysis.

We propose Macocha to mine version-control systems in order to detect novels change patterns. We used the k-nearest neighbor algorithm (KNN) to group changes into change periods and therefore to determine automatically the duration of the different change periods in each program. Then, we performed validations of Macocha on seven programs developed with three different languages: C, C++, and Java. We studied the variations in precision and recall of our approach when using different values of its parameters (the Hamming distance, the number s of

shifting profiles, and the start change periods). Finally, we provided evidence on the relevance of the Asynchrony and Dephase change patterns.

We are currently (1) relating change patterns with design patterns, (2) identifying other scenarios in which Asynchrony and Dephase change patterns help in reducing maintenance costs, (3) evaluating the consistency and the usefulness of change patterns' occurrences, including files recently changed over other occurrences, (4) relating Asynchrony and Dephase change patterns to program quality and external software characteristics, such as change proneness. Future work also includes empirical studies of the usefulness for developers of ranking occurrences of the Asynchrony and Dephase change patterns as well as applying Macocha to different C/C++ programs.

In next chapters, we will use Macocha to analyse the evolution and the co-evolution dependencies in programs and to conduct a study in order to detect the impact of the anti-patterns dependencies. Thus, we will focus on classes and on anti-patterns in object-oriented programs.

CHAPTER 4

RELATIONS BETWEEN EVOLUTION AND CO-EVOLUTION DEPENDENCIES AND FAULT-PRONENESS

4.1 Introduction

Several fault prediction approaches were proposed to analyse fault-proneness. While some approaches predict the presence or absence of faults for each component (the classification scenario), others predict the amount of faults affecting each component in the future, producing a ranked list of components. On the one hand, Change-Log Approaches [43] use process metrics extracted from the versioning system, assuming that recently or frequently changed classes are the most probable source of faults. On the other hand, Code-Metrics approaches [73] use source code metrics, assuming that complex or larger classes are more fault-prone. However, it is not clear how classes with different evolution behavior are linked with faults. Indeed, evolution studies did not link different evolution behavior to faults.

Two major kinds of class evolution dependencies are class lifetime and class co-evolution. For example, in ArgoUML, we spotted that hundreds of classes existed only during some program versions. We found that some of these classes, such as `GoModelToCollaboration` and `UMLInstanceClassifierListModel`, were created by developers to examine a feature which was later abandoned. We differentiate between classes that appear and disappear many times during the program lifetime, (Transient classes) and classes that appear only during one version of the program (Short-lived classes). Similarly, distinguishing between co-evolving classes (classes which exhibit similar evolution profiles, due to interdependencies among them [105]) and independently evolving classes could make a difference. For example, in XercesJ, we found that the class `DocumentBuilder` co-evolved with the class `SAXParser.java` from Xerces1.0.1 to Xerces2.0.0. Indeed, these two classes are related to the same fault fixed on 26th September 2002.

We present a novel approach, Profilo, to (1) group classes in an object-oriented programs according to their evolutionary histories, (2) spot their co-evolution dependencies, and (3) relate their evolution and co-evolution dependencies with fault-

proneness. Our goal is to spot the impact of maintenance activities and evolution dependencies on fault-proneness.

We apply our approach on three open-source programs: ArgoUML, JFreeChart, and XercesJ to answer the following research questions:

- **RQ5**: *RQ5: What is the relation between class lifetime and fault-proneness?*

  We decided to consider three types of class evolution: Persistent (classes that never disappear after their first introduction into the program), Short-lived (classes that appear only during one version of the program), and Transient classes (classes that appear and disappear many times during the program lifetime). We showed that Persistent classes are significantly less fault-prone than Short-lived and Transient classes.

- **RQ6**: *RQ6: What is the relation between co-evolution dependencies and fault-proneness?*

  We found that, in most cases, fixing faults in class A requires changing the co-evolved classes of A.

This chapter is organised as follows: Section 4.2 presents our approach Profilo. Section 4.3 describes our empirical study. Section 4.5 and Section 4.6 report and discuss its results as well as threats to its validity. Section 4.7 concludes with lessons learned and future work.

4.2 Approach: Profilo

This section presents our approach, Profilo, to analyse the link between program evolution and fault proneness. We will describe each step of the approach in details below as shown in Figure 4.1. Given several versions of an object-oriented program, Profilo extracts their class diagrams using an existing tool PADL[1] and creates the set of version-profiles that spots for each version all of its classes. Profilo identifies class renamings, class changes, and fault fixing using two approaches: ADvISE [40] and Macocha (Chapter 3). Profilo creates the set of class-profiles that describes

[1] `http://www.ptidej.net/tool/`

the evolution of each class in the program. Based on this set, it groups classes according to their co-evolution relations.

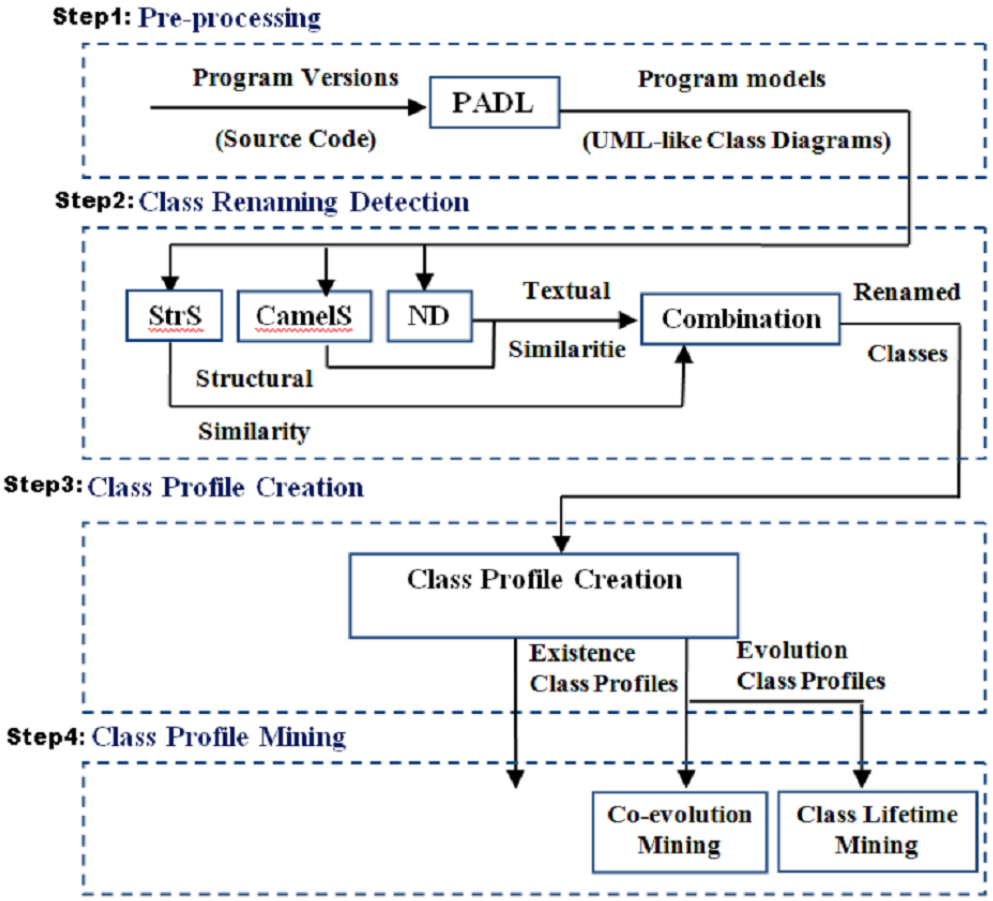

Figure 4.1 – Profilo Overview.

Step 1: Pre-processing

Profilo use PADL [35] to automatically reverse-engineer class diagrams from the source code of object-oriented programs[2]. PADL creates a meta-model to specify the source code and parses this meta model to detect all of the constituents found in any object-oriented system: class, interface, member class and interface,

[2]We consider six types of static relationships among classes: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations [34]

method, field, inheritance and implementation. The PADL tool is associated with several parsers to build models of software from AOL, C++, C#, and Java. Profilo also uses Macocha to identify the set of changes performed on each class by mining version-control systems. We compute the fault-proneness of a class by relating fault reports and commits to the class. Fault fixing changes are documented in text reports that describe different kinds of problems in a program. Thus, we trace faults/issues to changes by matching their IDs and their dates in the commits and in the bug reports. For example, we detect around three thousands classes in JFreeChart and we trace 420 faults.

Step 2: Class Renaming Detection

ADvISE identifies class renamings using the structure-based and the text-based metrics, which assess the similarities between original and renamed classes, as follows:

Step 2.1: Structural Similarity

ADvISE defines a structure-based similarity, $StrS$, between a candidate original class C_A (in version V_i) and a candidate renamed class C_B (in version V_{i+1}), as the percentage of their common methods, attribute types, and relations. We assume 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[3], and relations of C_A (respectively, C_B). 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 classes C_A and C_B do not have any common methods, attribute types, or relations. If $StrS(C_A, C_B) = 1$, then $S(C_A)$ and $S(C_B)$ are equal, *i.e.*, classes C_A and C_B have the same sets of methods, attribute types, and relations. Given, a class C_A, ADvISE reports the class C_B with the highest $StrS$ similarity as the best candidate renamed from C_A.

[3]For the sake of simplicity, we are interested in attribute types instead of attribute names. Because, attribute names could change between two versions (V_i and V_{i+1}).

ADvISE method is inspired by the Jaccard coefficient to quantify similarity between the sample sets $S(C_A)$ and $S(C_B)$. The Jaccard coefficient is a measure that provides a percentage of similarity of two sample sets, defined as the size of the intersection divided by the size of the union of the sample sets. As a set-based similarity measure, the Jaccard similarity coefficient does not discriminate between the set items. Justification for its use is primarily empirical rather than theoretical. On the one hand, instead of dividing the size of the intersection by the size of the union, ADvISE divides it by the size of the merge. That is, the items shared by two sample sets appear twice in our resulting set $Merge(X, Y) = \{a, a, b\}$. On the other hand, ADvISE similarity ranges between zero and one, like Jaccard.

Step 2.2: Textual Similarity

Given an original class C_A, ADvISE reports a set of best candidate renamed classes $\{C_{B_1}, ..., C_{B_n}\}$ that have the highest $StrS$ similarity. We want to select the 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 types, methods and relations. To reinforce $StrS$, ADvISE computes the textual similarity between the name of the original class C_A and the name of 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).

ADvISE computes $CamelS$ similarity between C_A and C_B as the percentage of common tokens between the name of C_A and the name of 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). ADvISE computes 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.

The Levenshtein Distance[68] 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. For example, the Lev-

enshtein distance between *Saturday* and *Sunday* is 3, because *Sunday* is obtained from *Saturday* by removing two characters *at* and substituting one from *r* to *n*. To have comparable Levenshtein distances, ADvISE uses the normalised edit distance (*ND*), given by

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

Step 2.3: Combination of Similarities

ADvISE combines *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.

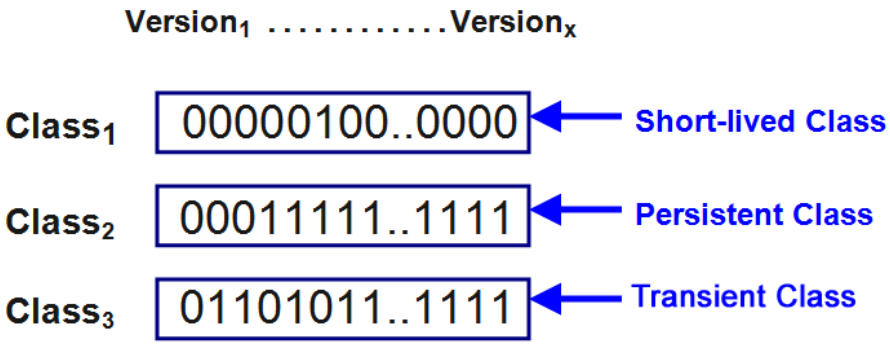

Figure 4.2 – Types of class evolution considered in this study.

Step 3: Class-profiles Creation

Profilo mines source code and version control systems to create a class-profile for each class. Then, we use this class-profile to extract the co-evolution relations among classes. It is defined as a vector $y = y_1 ... y_m$, where m represents the number of versions. The value of y_i indicates whether the class C is present, renamed,

changed, or deleted in the i^{th} version.

$$y_i = \begin{cases} 1 & \text{if class is present at version } i \\ 0 & \text{otherwise.} \end{cases}$$

Step 4: Class-profiles Mining

Mining Class Lifetime: We classify classes according to their class-profiles. Then, Profilo reports three types of class evolution as shown in Figure 4.2.

Short-lived classes: They have a very short lifetime, *i.e.*, they exist only during one version of the program. Such classes may have been created to try out an idea that was then dropped or modified.

Persistent classes: These classes are never deleted after their first introduction. On the one hand, Persistent classes should be examined, as they may represent cases of dead code that no developer dares to remove as there is no one being able to explain the purpose of these classes. On the other hand, Persistent classes may be considered to be part of a tunnel [59], the backbone part of the program, as they have not been removed since their first appearance in a given version of a program. Hence, we also mine version control systems to assess whether a Persistent class is dead code or not.

Transient classes: They appear and disappear many times during the program lifetime. Such classes may have been involved in many design choices and should be analysed, as they represent cases of design decision changes.

Mining Co-evolution Dependencies: We group classes that have the same Evolution Class-profile and are related by static relationships. Such classes are added, renamed, changed, and could be deleted in the same versions. They are related, also, by static relationships (use, association, aggregation, and composition relationships). Detecting co-evolution among numerous classes in the program help developers in assessing their effort to implement change requests and in performing the most adequate changes.

For example, in JFreeChart, we found that the period with the highest number of faults found and fixed for this project was before the publication of the version34 (jfreechart-1.0.2) in August 2006 and in the period of publication of the version37 (jfreechart-1.0.5) in March 2007.

On the other hand, the majority of Transient and Short-lived classes in JFreeChart were added in this period (more than 70% of Transient and Short-lived classes on this program). We suspect a correlation, between the introducing of these classes in the program and the increasing number of faults on this period. At the same time, some of this classes are added, renamed, and changed on the same version over their whole lifespan. We found that these classes have similar evolution trends and that many of them are involved in the same faults. Detecting dependencies of evolution of these classes could explain and possibly prevent faults by being sure that changes are propagated adequately by developers among them.

4.3 Empirical Study

Following the Goal Question Metric (GQM) [8], the *goal* of this study is (1) to detect interesting observations on the relationship between the evolution of object-oriented source code at class level and fault-proneness, (2) to detect co-evolution dependencies to explain and possibly prevent faults, and (3) to confirm these observations statistically. The *quality focus* is the reduction of comprehension cost and maintenance effort. The *perspective* is of both researchers, who want to study the relationship between program evolution and fault-proneness, and practitioners, who analyse software evolution to estimate the effort required for future maintenance tasks. The *context* of our experiment is three open-source Java programs: ArgoUML, JFreeChart, and XercesJ.

4.3.1 Objects

We apply our approach on three Java programs: ArgoUML[4], JFreeChart[5], and XercesJ[6]. We use these programs because they are open source, have been used in previous work, are of different domains, span several years and versions, and underwent between thousands and hundreds of classes. Table 4.1 summarises some statistics about these programs.

[4] `http://argouml.tigris.org/`
[5] `http://www.jfree.org/`
[6] `http://xerces.apache.org/xerces-j/`

We analyse the evolution of ArgoUML program for a period of nine years, from 2002-10-09 to 2011-04-03. In this period, ArgoUML has gone through over 18 major versions, from the version 0.10.1, to the version 0.32.2.

We analyse the evolution of JFreeChart for a period of 10 years. In this period, JFreeChart has gone through 46 major versions, from the first published version on December 2000 to version 1.0.14 on November 2011.

We analyse the evolution of XercesJ for a period of three years, from 2003-10-13 to 2006-11-23. In this period, XercesJ has gone through 36 major versions.

	ArgoUML	JFreeChart	XercesJ
Versions	18	46	36
Start study	02-10-09	00-12-01	03-10-13
End study	11-04-03	11-11-20	06-11-23
From Version	0.10.1	0.5.6	1.0.1
To Version	0.32.2	1.0.13	2.9.0
# of classes	2011	1938	892
# of Faults fixed	218	130	34

Table 4.1 – Descriptive statistics of the object programs.

As in previous work [55], fault-proneness refers to whether a class underwent at least a fault fixing change during the study periods. Fault fixing changes are documented in text reports that describe different kinds of problems in a program. They are usually posted in issue-tracking systems *e.g.*, Bugzilla, for the three studied programs by users and developers to warn their community of pending issues with its functionalities; issues in these systems deal with different kinds of change requests: fixing faults, restructuring, and so on. We trace faults/issues to changes by matching their IDs in the commits and by manual validation.

4.4 Exploratory Study Analyses

In essence exploratory studies are undertaken to better comprehend the link between program evolution and fault-proneness since very few studies might have been considered in that area.

We use data collected in the three programs and from externals information to discuss typical examples as follows:

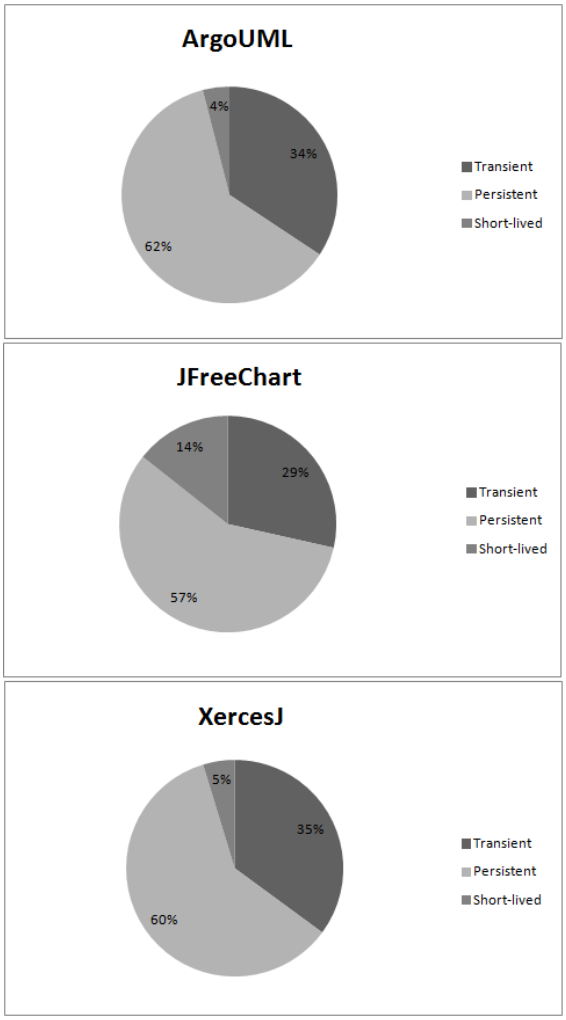

Figure 4.3 – Distribution of class lifetimes detected by Profilo.

	ArgoUML	JFreeChart	XercesJ
Transient	690	645	313
Persistent	1241	1293	537
Short-lived	80	324	42
# of Co-Evolution	42	11	23

Table 4.2 – Cardinalities of the sets obtained in the study.

Persistent classes. In Figure 4.3, we note that most classes in ArgoUML, JFreeChat, and XercesJ are Persistent (more than 60% of classes). On the one

hand, these classes represent the stable backbone (tunnel) of the program such as `org.argouml.uml.generator.ui.ClassGenerationDialog` in ArgoUML. In fact, this class implements the java code generator in this program and was maintained 82 times by several developers (tfmorris, penyaskito, mvw, etc.). On the other hand, Persistent classes could represent also dead code, such as `SDNotationSettings`. Indeed, this class was never changed after its introduction in ArgoUML on March 1999 by tfmorris. We noted that in ArgoUML, more than 80% of classes were maintained three times at most. On the other hand, less than 1% of classes were maintained 50 times at least. We observed also 218 fault fixes in this system from 2002-10-09 to 2011-04-03.

Transient classes. We detect classes that appear and disappear many times during the maintenance of the three programs. For example, the class `OverlaidCategoryPlot`, appeared in JFreeChart in the version 0.9.9 in June 2003, and was deleted in the version 0.20.0 before reappeared in the next version. In fact, developers detect faults in this class, and that explain the Nonpersistence of this class. For example, in the Bugzilla of JFreeChart, the bug ID576760[7] reported in relation with this class that "No outline for overlaid category plot" when developers used category plots in one application.

Short-lived classes. They represent the smallest group of classes in the three analysed programs. Such classes were created to try out an idea that was then dropped or modified, or to test some program behavior. For example, the class `org.jfree.chart.demo.TimePeriodToStringTest` was created in JFreeChart0.9.9 published in July 2003 to test information encapsulated in `TimePeriod` in order to fix a fault[8] related to this class. After this version, this class was deleted.

Co-evolution. The development and maintenance of a program involves handling a large number of classes. Knowing that two or more classes follow the same co-evolution pattern helps developers to maintain properly the dependencies between these classes in the program. Otherwise, they lead to faults in the program. For example, in JFreeChart, we find that `ChartPanel` and `CombinedDomainXYPlot`. were introduced, changed and renamed in the same versions but in different periods

[7] `http://sourceforge.net/tracker/index.php?func=detail&aid=576760&group_id=15494&atid=115494`

[8] `http://sourceforge.net/tracker/index.php?func=detail&aid=814424&group_id=15494&atid=365494`

and by different developers. Thus, co-change analysis cannot report their dependency. Profilo reports that these two classes co-evolved and the bug ID1950037[9] reported "a bug either in ChartPanel or CombinedDomainXYPlot when trying to zoom in/out on the range axis" and confirmed the dependency between these two classes.

4.4.1 Research Questions

We break down our study into two steps and we seek answers to the following two research questions:

- ***RQ5: What is the relation between class lifetime and fault-proneness?***
- ***RQ6: What is the relation between co-evolution dependencies and fault-proneness?***

These questions investigate the impact of the evolution of different artefacts of a program on fault-proneness.

4.5 Study Results

Table 4.2 summarises the results obtained by applying Profilo. We validated Profilo results manually and checked external sources of information provided by bugs reports, mailing lists, and requirement descriptions to confirm and to discuss results. The analysis reported in this section have been performed using the R statistical environment[10]. We use the contingency tables to assess the direction of the difference of fault proneness across different groups of classes. In statistics, a contingency table is a table in a matrix format that displays the frequency distribution of the variables. Although in practice it is employed when sample sizes are small, it is valid for all sample sizes.

[9] `http://sourceforge.net/tracker/index.php?func=detail&aid=1950037&group_id=15494&atid=115494`

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

4.5.1 *RQ5: What is the relation between class lifetime and fault-proneness?*

4.5.1.1 Motivation

We group classes according to their profiles through the program lifespan, taking into consideration the renaming, refactoring, and structural changes of classes, to determine how class lifetime models are related to fault-proneness.

4.5.1.2 Method

We use Fisher's exact test [90] to check whether the difference is significant in order to assess the direction of the difference of fault proneness across different groups of classes.

Fisher's exact test [90] is a statistical significant test used in the analysis of contingency tables.

The test is useful for categorical data that result from classifying objects in two different ways. It is used to examine the significance of the association (contingency) between the two kinds of classification, in our study: Faulty classes and clean classes.

To compute the p-value of the test, the contingency tables must then be ordered by some criterion that measures dependence and those tables, that represent equal or greater deviation from independence than the observed table, are the ones whose probabilities are added together. The contingency tables tested in this study contain the total numbers of faulty and clean classes identified in ArgoUML, JFreeChart, and XercesJ.

We also compute the odds ratio [90] that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the odds that Short-lived and Transient classes are identified as fault-prone, to the odds q of the same event occurring in the other sample, *i.e.*, the odds that Persistent classes are identified as fault-prone. An odds ratio greater than 1 indicates that the event is more likely in the first sample, while an odds ratio less than 1 that it is more likely in the second sample. An odds ratio $OR = \frac{p/(1-p)}{q/(1-q)}$.

$OR > 1$ indicates that fault-prone entities have high term entropy and high context coverage. We expect $OR > 1$ and a statistically significant p-value.

We verify the null hypothesis that we state as:

- H_{RQ5_0}: There is no statistically significant difference between proportions of faults carried by Persistent, Short-lived, and Transient classes in ArgoUML, JFreeChart, and XercesJ.

If we reject the null hypothesis H_{RQ5_0}, then we explain the rejection either as:

- H_{RQ5_1}: There is a statistically significant difference between proportions of faults carried by Persistent, Short-lived and Transient classes.

To attempt rejecting H_{RQ5_0}, we test whether the proportion of classes in ArgoUML, JFreeChart, and XercesJ that compose Short-lived and Transient (respectively Persistent) classes take part (or not) in significantly more faults than those in Persistent (respectively Short-lived and Transient) classes. We merged Short-lived and Transient classes because our goal is to help developers to prevent faults and, for future releases, a Short-lived class could became a Transient class and a Transient class could became Short-lived.

4.5.1.3 Results

Table 4.3 presents a contingency table for ArgoUML, JFreeChart and XercesJ that reports the number of (1) Short-lived and Transient classes that are identified as fault-prone; (2) Short-lived and Transient classes that are identified as clean; (3) Persistent classes that are identified as fault-prone; and, (4) Persistent classes that are identified as clean. The result of Fisher's exact test and odds ratios when testing H_{RQ5_0} are significant. In Table 4.3, the p-value is less then 0.05 and the odds ratio for fault-prone Short-lived and Transient classes is two times higher than for fault-prone Persistent classes.

We can answer to **RQ5** as follows: we showed that Persistent classes are significantly less fault-prone than Short-lived and Transient classes.

	Faulty	Clean
ArgoUML's Non-Persistent classes	400	370
ArgoUML's Persistent classes	326	915
JFreeChart's Non-Persistent classes	312	657
JFreeChart's Persistent classes	366	927
XercesJ's Non-Persistent classes	268	277
XercesJ's Persistent classes	170	508
The Sum of Non-Persistent classes	980	1304
The Sum of Persistent classes	862	2350
Fisher's test	2.2e-16	
Odd-ratio	2.048582	

Table 4.3 – Contingency table and Fisher test results in ArgoUML, JFreeChart and XercesJ for Persistent, Non-Persistent classes (Short-lived and Transient classes) with at least one fault.

4.5.2 *RQ6: What is the relation between co-evolution dependencies and fault-proneness?*

4.5.2.1 Motivation

The goal of analysing dependencies among co-evolved classes (clusters of classes exhibit similar evolution profiles) is to check if the proportion of faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes.

4.5.2.2 Method

The Chi-Square test is used to test the different proportions of faults fixed for co-evolved and not co-evolved classes. Indeed, the Chi-Square statistic is used to investigate whether distributions of categorical variables differ from one another.

We test for statistical significance to verify the null hypothesis that we state as:

- H_{RQ6_0}: There are no statistically significant between proportions of faults involving co-evolved classes or not co-evolved classes in the three programs.

If we reject the null hypothesis H_{RQ6_0}, then we explain the rejection either as:

- H_{RQ6_1}: The proportion of faults carried by co-evolved classes is not the same as the proportion of faults carried by not co-evolved classes.

To attempt rejecting H_{RQ6_0} we test whether the proportion of co-evolved classes in ArgoUML, JFreeChart and XercesJ take part (or not) in significantly more faults than other classes.

4.5.2.3 Results

We use in this test the contingency Table 4.4, where rows represent the number of faults involving co-evolved classes and the number of faults involving non co-evolved classes. The result of Chi-Square test and odds ratios when testing H_{RQ6_0} are significant. The p-value is less then 0.05 and we can reject the null hypothesis.

	Faults involving CC	Faults involving NCC
ArgoUML	126	92
JFreeChart	69	61
XercesJ	19	15
The p-value of Chi-Square	0.01859	

Table 4.4 – Contingency table and Chi-Square test results in ArgoUML, JFreeChart and XercesJ for faults fixed by co-evolved (CC)or not co-evolved classes (NCC).

We can answer to **RQ6** as follows: faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes.

4.6 Discussions

4.6.1 Class Lifetime and Fault-proneness

In this thesis, we combine information obtained from class evolutionary history and from bug reports to obtain a clearer picture of the evolution of object-oriented program. This is a key knowledge for a maintenance activity, because it allows us to detect the critical parts of the program that represent the starting point for a maintenance process. We found that Non-Persistent classes should be spotted

and well-understood before maintaining the programs as these classes are more fault-prone. Special attention must be given to these entities to keep the design intact during program evolution because the instability of these classes could have a negative impact on the fault-proneness of the program.

4.6.2 Similarities in Classes Evolution Profiles

While co-change dependencies analysis reports the sets of classes that are often changed together, our approach reports the sets of classes that evolve in parallel ways and not necessarily at the same time. To the best of our knowledge, previous co-change approaches did not use method such as structure-based and text-based similarities to identify class renamings and, therefore, they could not report co-change or co-evolution relations among renamed classes. In this thesis, we noted that such relations describe implicit design dependencies and source code evolution. Thus, special attention must be given to these relations to keep the design intact during program maintenance activity. If numerous co-evolution relations exist, Profilo sorts the sets of results depending on the number of static relationships among co-evolved classes in order to help the developers to focus on those that potentially led to a design flaw or to mistakes in maintaining classes together.

4.6.3 Threats to Validity

Some threats limit the validity of our empirical study.

Construct Validity. Construct validity threats concern the relation between theory and observations. In this study, they could be due to the errors of the implementation. They could also be due to an imprecision of our measurements of the distance between different couples of class-profile and–or different couples of version-profile. We believe that this threat is mitigated by the facts that we validated Profilo results using external sources of information (bug reports and others).

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, in **RQ5** and in **RQ6**. We cannot claim causation, but our discussion tries to explain why some classes could have been subject to faults.

Reliability Validity. Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to re-implement our approach and replicate our empirical study. The programs, change logs, and raw data to obtain our observations are available online at `http://www.ptidej.net/downloads/experiments/csmr13a/`.

External Validity. External Validity concern the generalisation of our findings. Although we performed our analyses on three different programs, belonging to different domains and with different sizes, we are aware that further empirical validations on a larger set of programs would be beneficial to better support our findings. We cannot assert that our results and observations are generalisable to any other program and the fact that all the analysed programs are open source and are developed with Java may reduce this generability.

4.7 Summary and Lessons Learned

In this chapter, we described a novel approach to analyse evolution and co-evolution dependencies and to trace fault-proneness. One of the goals addressed in this thesis is how we can relate dependencies of classes in object-oriented programs with fault-proneness. The concepts of class lifetime and co-evolution helped us to describe and to identify the reasons that have driven the programs' codes to their current states. We showed that Persistent classes are significantly less fault-prone than other classes and that faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes. Profilo draws, also, informed conclusions about the relation between maintenance tasks and fault-proneness in order to help developers to understand evolution trends and to maintain the programs correctly.

Future work aims at (1) analysing further co-evolution relations by replicating our study with other larger programs, (2) performing a comprehensive study of the relationships between class lifetime and change-proneness, and (3) identifying the lifetime followed by design motifs such as design patterns and anti-patterns.

In the next chapter, we describe an empirical study, performed on three object-oriented systems, to analyse the impact of anti-patterns dependencies on fault-

proneness. We also use Padl tool to detect anti-patterns static relationships and Macocha anti-patterns co-changes dependencies.

CHAPTER 5

RELATIONS BETWEEN ANTI-PATTERNS DEPENDENCIES AND FAULT-PRONENESS

5.1 Introduction

Software systems are never complete and evolve continuously [67]. As they evolve, their complexity grows. Prior work has shown that software complexity is an obstacle to introducing changes and that complex modules tend to be fault-prone [61, 65]. Developers often introduce bad solutions, anti-patterns [102], to recurring design problems in their systems and these anti-patterns lead to negative effects on code quality.

In the same context, anti-patterns are known as motifs that are usually thought to be good solutions to some design or implementation problems, but back-fires badly when applied.

While existing work has shown that anti-patterns are problematic ([96], [86], and [55]), we believe that more attention should be focus on static and co-change relationships between anti-patterns classes and other classes without anti-patterns. We conjecture that, static and co-change relationships with anti-patterns can impact the fault-proneness classes without anti-patterns. A recent finding by Radu and Cristina Marinescu [71] that clients of classes with Identity Disharmonies are more fault-prone than other classes, supports this conjecture. The static relationships between anti-patterns classes and other classes (and vice versa) are typically use, association, aggregation, and composition relationships [34]. Also, classes participating in anti-patterns may have "hidden", temporal dependencies. These dependencies occur when developers know that, when changing a class, they must also change another. The literature describes many approaches to extract and analyse such hidden dependencies and to infer the patterns that describe these changes to help developers to maintain their systems. For example, some previous work [15, 115] detected motifs that highlight co-changing groups of classes and that describe the (often implicit) dependencies or logical couplings among classes that have been observed to frequently change together [31]. Two classes are co-

changing if they were changed by the same author and with the same log message in a time-window between some milliseconds and some minutes at the most [31], [115]. Recently, we introduced the novel concept of macro co-change[1] and proposed detection algorithms to identify various co-change situations among the classes of a software system[47].

In this Chapter, we analyze static and temporal relationships (*i.e.*, co-changes) between anti-pattern and (non)anti-pattern classes from three Java open source software systems: ArgoUML, JFreeChart, and XercesJ.

Research Problem

On one hand, previous work agree that anti-patterns are commonly introduced by developers but they are more fault prone and counterproductive in program development and maintenance [86]. On the other hand, static relationships and co-change dependencies can be "channels" propagating faults among classes in software systems. However, there is no much information available in the literature about the fault proneness of classes having static or co-change dependencies with classes infected by anti-patterns. In this study, we are looking for evidence that practitioners should pay attention to systems with a high number of classes related to classes infected by anti-patterns, because these classes are likely to be the subject of their change efforts.

As in previous work [6], we assume that a class C co-changes with the anti-pattern A if C co-changes at least with one class belonging to A. We also assume that a class S has a static relationships with the anti-pattern A if S has a use, association, aggregation, or composition relationships with at least one class belonging to A in one of the versions of the analysed systems.

We analyse dependencies with anti-patterns in two ways: first, we investigate whether classes having static relationships (use, association, aggregation, and composition relationships) with anti-patterns classes are more fault-prone than others. Second, we investigate whether classes co-changing with anti-patterns classes are more fault-prone than others. We formulate the following research questions:

[1] two or more changed files that exactly change together with long time intervals between their changes and–or performed by different developers and with different log messages

- **RQ7:** RQ7: Are classes that have static relationships with anti-patterns more fault-prone than other classes?
- **RQ8:** RQ8: Are classes that co-change with anti-patterns more fault-prone than other classes?

We found that, in ArgoUML, JFreeChart, and XercesJ, classes having static or co-change dependencies with anti-patterns are more fault prone. We also found that such dependencies can be used to predict fault and–or improve fault prediction models.

Organisation

Section 5.2 presents our approach. Section 5.3 describes our empirical study. Section 5.4 presents the study results while Section 5.5 discusses them along with threats to their validity. Finally, Section 5.6 concludes the study and outlines future work.

5.2 Approach: AntImpacts

This section describes the steps necessary to extract and analyse the data required to perform this study.

Step 1: Extracting Anti-patterns From the Source Code

We use the DEtection for CORrection approach DECOR [96], to specify and detect anti-patterns. DECOR is based on a thorough domain analysis of anti-patterns defined in the literature and provides a domain-specific language to specify code smells and anti-patterns and methods to detect their occurrences automatically. It can be applied on any object-oriented system through the use of the PADL [35] meta-model and POM framework [36]. PADL describes the structure of systems and a subset of their behavior, *i.e.* classes and their relationships. POM is a PADL-based framework that implements more than 60 structural metrics.

Indeed, DECOR proposes a domain-specific language to specify and generate automatically design defect detection algorithms. A domain-specific language offers greater flexibility than ad hoc algorithms because the domain experts, the software engineers, can specify and modify manually the detection rules using high-level

abstractions, taking into account the context, environment, and characteristics of the analysed systems. Moreover, the language allows specifying defect detection algorithms at a high-level of abstraction using key concepts found in their text-based descriptions.

We use seven of these metrics to verify if we find differences in fault-proneness between classes having dependencies with anti-patterns and other classes with similar complexity or size. These metrics measure : (1) the total lines of code per class; (2) the number of method calls of a class; (3) the nested block depth of the methods in a class; (4) the number of parameters of the methods in class; (5) the McCabe cyclomatic complexity of the methods in a class; (6) the number of fields of a classes; and (7) the number of methods of a classes. We choose these seven metrics because they have been successfully used to predict post-release faults [74].

We parse the CVS change logs of our subject systems and apply the heuristics by Sliwersky *et al.* [93] to identify fault fix locations. Precisely, we parse commit log messages using a Perl script and extract bug IDs and specific keywords, such as "fixed" or "bug" to identify fault fixing commits. For each fault fixing commit, we extract the list of files that were changed to fix the fault.

Step 2: Detecting Anti-patterns Static Relationships

We use the Ptidej tool suite [35] to detect anti-patterns static relationships. Ptidej characterizes the constituents of class diagrams and proposes algorithms to identify these constituents in source code. Ptidej distinguishes use, creation, association, aggregation, and composition relationships because such relationships exist in most notations used to model systems. This approach uses the PADL [35] meta-model and parses the source code of systems to detect models that include all of the constituents found in any object-oriented system: class, interface, member class and interface, method, field, inheritance and implementation relationships, and rules controlling their interactions. Ptidej depends on a set of definitions for unidirectional binary class relationships that they are proposed and formalized in a previous work [35].

The formalizations define the relationships in terms of four language-independent properties that are derivable from static and dynamic analyses of systems: exclusivity, type of message receiver, lifetime, and multiplicity.

Step 3: Detecting Anti-pattern Temporal Dependencies

We use our approach Macocha [47] to mine software repositories and identify classes that are co-changing with anti-patterns. Macocha mines version-control systems (CVS or SVN) to identify, among changed classes, those that are co-changing with anti-patterns.

Macocha also calculates the following process metrics, defined and successfully used in previous work [114] to predict software faults. These metrics are used to verify if we find a difference in fault-proneness between classes having dependencies with anti-patterns and other classes. Thus, process metrics are used to check if classes having similar change histories are more or less fault-prone than classes having dependencies with anti-patterns. Indeed, Macocha identifies fixes, for example, in version archives as follow: within the messages that describe changes, Macocha search for references to bug reports such as "fixed" or "bug" or matches patterns like " # and a number". Macocha match faults/issues with changes by matching their IDs in the commits. Here are the process metrics calculated with the Macocha approach as defined in [114]:

1. Total Prior Changes: measures the total number of changes to a class in the 6 months period before the release.

2. Prior Fault Fixing Changes: the number of fault fixing changes done to a class in the 6 months period before the release.

3. Pre-release faults: the number of pre-release faults in a class in the 6 months period before the release (these are faults observed during development and testing of a program).

4. Post-release faults: the number of post-release faults in a class in the 6 months period after the release (these faults are observed after the program has been deployed to the users).

Step 4: Analysing Anti-patterns Dependencies

Table 5.3 provides some statistics about the anti-patterns found in the subject systems considered in this Chapter. To perform the empirical study, we choose to analyse the relationships of well known anti-patterns. We choose these anti-patterns because they are representative of problems with data, complexity, size,

and the features provided by classes [55]. We also use these anti-patterns because they have been used and analysed in previous work [55], [96].

Fault-proneness refers to whether a class underwent at least one fault fixing in the system life cycle [55]. Fault fixings are documented in bug reports that describe different kinds of problems in a system. They are usually posted in issue-tracking systems, *e.g.*, Bugzilla for the three studied systems, by users and developers to warn their community of pending issues with its functionalities.

In **RQ1**, we test whether the proportion of classes in ArgoUML, JFreeChart, and XercesJ that have static relationships with anti-patterns classes have (or do not have) significantly more faults than those that do not have static relationships with anti-patterns classes.

In **RQ2**, we test whether the proportion of co-changed classes with anti-patterns in ArgoUML, JFreeChart, and XercesJ have (or do not have) significantly more faults than the other classes.

Because previous studies [41, 114] have found size, complexity and process metrics to be good predictors of faults in software systems. We perform an experiment to verify if static relationships and–or co-change relations can provide additional information over these traditional fault prediction metrics. Precisely, our experiment consists in building two models for predicting the presence or absence of faults in classes: (1) one using only change and code metrics and (2) one using change metrics, code metrics, and anti-pattern dependencies information. In our experiment, the independent variables are the collection of code and process metrics and the dependent variable is a two value variable that represents whether or not a class has one or more post-release fault. There are various machine learning methods available to build such models. We use Support Vector Machines to build the prediction models because this machine learning method has been widely used in literature and has shown good results [77], [73]. The models output the likelihood of a class to have one or more post release faults. We use statistical tests to examine (the significance of) the difference between the performance of the two models when predicting faults. More specifically, we use off-the-shelf methods

from the R[2] statical package to analyze the statistical significance and collinearity attributes of the independent variables used in our experiment.

Classes belonging to an anti-pattern can have dependencies (static relationships and–or co-change dependencies) with classes belonging to other anti-patterns. Thus, the tests reported in this Chapter cover classes that have a dependency with an anti-pattern, regardless of the fact that these classes could belong to other anti-patterns. Nevertheless, we present in Section 5.4 the result of our analysis of the impact of anti–patterns dependencies, for classes belonging to anti-patterns and other classes separately.

5.3 Empirical Study

The *goal* of our study is to assess whether classes having dependencies with anti-patterns have a higher likelihood than other classes to be involved in issues documenting faults. The *quality focus* is the improving of program comprehension and the reducing of maintenance effort by detecting and using anti-patterns static or co-change dependencies. The *context* of our study is both the comprehension and the maintenance of systems.

	ArgoUML	JFreeChart	XercesJ
# of classes	3,325	1,615	1,191
# of snapshots	4,480	2,010	159,196
# of AntiSingleton	3	38	24
# of Blob	100	49	12
# of ClassDataShouldBePrivate	51	3	6
# of ComplexClass	158	52	7
# of LongMethod	336	75	7
# of LongParameterList	281	76	4
# of MessageChains	162	59	8
# of RefusedParentBequest	123	5	7
# of SpaghettiCode	1	2	6
# of SpeculativeGenerality	22	3	29
# of SwissArmyKnife	13	26	29

Table 5.1 – Descriptive statistics of the object systems.

[2]http://www.r-project.org/

5.3.1 Research Questions

We break down our study into two steps:

- **RQ7:** RQ7: Are classes that have static relationships with anti-patterns more fault-prone than other classes?

First, we check if classes having static relationships (use, association, aggregation, and composition relationships) with anti-patterns classes are more fault-prone than others classes in the three analysed programs.

- **RQ8:** RQ8: Are classes that co-change with anti-patterns more fault-prone than other classes?

Second, we investigate whether classes that are co-changing with anti-patterns classes are more fault-prone than others classes.

We test the two null hypotheses state:

- H_{RQ7_0}: The proportions of faults carried by classes having static relationships with anti-patterns and other classes are the same in the programs.
- H_{RQ8_0}: The proportions of faults involving classes having co-change dependencies with anti-patterns and other classes are the same in the programs.

If we reject the null hypothese H_{RQ7_0}, it could mean that the proportions of faults carried by classes having static relationships with anti-patterns and faults carried by other classes in the analysed programs are not the same.

If we reject the null hypothese H_{RQ8_0}, we explain the rejection as that the proportion of faults carried by classes co-changing with anti-patterns is not the same as the proportion of faults carried by classes not co-changing with anti-patterns.

5.3.2 Objects

We apply our approach on three Java programs: ArgoUML[3], JFreeChart[4], and XercesJ[5]. We use these programs because they are open source, have been used in

[3] `http://argouml.tigris.org/`
[4] `http://www.jfree.org/`
[5] `http://xerces.apache.org/xerces-j/`

previous work, are of different domains, span several years and versions, and have between hundreds and thousands of classes. Table 5.3 summarises some statistics about these programs.

For anti-patterns dependencies analysis in ArgoUML, we extracted a total number of 4,480 snapshots in the time interval between September 27th, 2008 and December 15th, 2011.

For JFreeChart, we considered an interval of observation ranging from June 15th, 2007 (release 1.0.6) to November 20th, 2009 (release 1.0.13 ALPHA). In such interval we extracted 2,010 snapshots.

For anti-patterns dependencies analysis in XercesJ, we extracted a total number of 159,196 snapshots from release 1.0.4 to release 2.9.0 in the time interval between October 14th, 2003 and November 23th, 2006.

5.3.3 Analyses

The analysis reported in Section 5.4 have been performed using the R statistical environment[6]. We use Fisher's exact test [90] to check whether the difference is significant. We also compute the odds ratio [90] that indicates the likelihood for an event to occur. In this study, the odds ratio is defined as the ratio of the odds that classes having static relationships with anti-patterns are identified as fault-prone to the odds that the rest of classes are identified as fault-prone.

5.4 Study Results

We now present the results of our empirical study. Tables 5.4.2, 5.4.1 and 5.4.2 summarise our findings.

5.4.1 RQ7: Are classes that have static relationships with anti-patterns more fault-prone than other classes?

Table 5.4.1 reports for ArgoUML, JFreeChart, and XercesJ the numbers of (1) classes having static relationships with anti-patterns and identified as fault-prone;

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

(2) classes having static relationships with anti-patterns and identified as clean; (3) classes without static relationships with anti-patterns and identified as fault-prone; and, (4) classes without static relationships with anti-patterns and identified as clean. The result of Fisher's exact test and odds ratios when testing $HRQ7_0$ are significant for all three programs. The p-value is less then 0.05 and the odds ratio for fault-prone classes related to anti-patterns by static dependencies is two times higher for fault-prone than other classes in the three programs.

We can answer positively to **RQ7**: we showed that classes having static relationships with anti-patterns are significantly more fault-prone than other classes.

But: Two observations limit the results of **RQ7**: First, in the three programs, as showed in Table 5.4.2, we do not detect any class having static dependencies (use, association, aggregation, and composition relationships) with SpaghettiCode. In this case, we can not relate the impact of using this anti-pattern and the fault-proneness of other classes in the programs. Second, based on complexity metrics and change metrics analysis, it is neither possible to conclude that other classes having the same complexity, change history, and code size are less fault-prone than classes having static relationships with anti-patterns nor is the opposite true. In fact, we take as input the list of code and change metrics described in Section 5.2 and check if there are a significant statistical difference on fault proneness between a model based on only these metrics and a model based on these metrics plus anti-patterns static relationships. If all anti-patterns are considered in this comparison, it is impossible to definitely exclude the possibility that there is no statistically differences in fault-proneness between classes related to anti-patterns and other classes with the same complexity, change history, and code size. However, If we group the results according to distinct anti-patterns, we observe that classes having static relationships with Blob, ComplexClass, and SwissArmyKnife are significantly more fault prone than other classes with the same complexity, change history, and code size. Future work include the categorisation of anti-patterns according to the impact of their dependencies on fault proneness.

Other observations: Many anti-patterns static relationships were with classes playing roles in design patterns. Opposite to anti-patterns, design patterns [32] are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to

changes. As a consequence, these classes, playing roles in design patterns and having static relationships with anti-patterns, can bias the results. Indeed, our finding shows cases that developers wrapped anti-patterns using design patterns to facilitate maintenance tasks and reduce comprehension effort. For example, in XercesJ v1.0.4, the class `org.apache.xerces.validators.common.XMLValidator.java` is an excessively complex class interface. The developer attempted to provide for all possible uses of this class. In her attempt, she added a large number of interface signatures to meet all possible needs. The developer may not have a clear abstraction or purpose for `org.apache.xerces.validators.common.XMLValidator.java`, which is represented by the lack of focus in its interface. Thus, we claim that this class belongs to a SwissArmyKnife anti-pattern. This anti-pattern is problematic because the complicated interface is difficult for other developers to understand and obscures how the class is intended to be used, even in simple cases. Other consequences of this complexity include the difficulties of debugging, documentation, and maintenance. We detect that this class has a use-relationship with the class `org.apache.xerces.validators.dtd.DTDImporter.java`, which belongs to the Command design pattern. Using Command classes makes it easier to construct general components that delegate sequence or execute method calls at a time of their choosing without the need to know the owner of the method or the method parameters. Thus, developer can correct `org.apache.xerces.validators.common.XMLValidator.java`, by using the related Command pattern, to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method, and values for the method parameters. Thus, by using the relationships of an anti-pattern with a specific design pattern, we could help developers to maintain the anti-pattern classes while reducing its influence on the system by benefiting from its relationships with other design pattern so that, in the long term, developers could eliminate this anti-pattern while propagating changes adequately. We plan to study in future work the effect of knowing and using the relationships of anti-patterns and design patterns in maintenance tasks and comprehension effort.

	Faulty	Clean
Classes having S.R. with AP in ArgoUML	1062	1003
Other classes in ArgoUML	681	579
Classes having S.R. with AP in JFreeChart	432	226
Other classes in JFreeChart	310	647
Classes having S.R. with AP in XercesJ	445	121
Other classes in XercesJ	126	499
Total of classes related to AP	1939	1350
Total of other classes	1117	1725
The p-value of Fisher's test	$2.2e-16$	
Odd-ratio	2.21802	

Table 5.2 – Contingency table and Fisher test results in ArgoUML, JFreeChart and XercesJ for classes with at least one fault (S.R.: Static Relationships, AP: Anti-pattern.

5.4.2 RQ8: Are classes that co-change with anti-patterns more fault-prone than other classes?

Table 5.4.2 presents a contingency table for ArgoUML, JFreeChart, and XercesJ that reports the number of (1) classes co-changing with anti-patterns and identified as fault-prone; (2) classes co-changing with anti-patterns and identified as clean; (3) other classes identified as fault-prone; and, (4) other classes identified as clean. The result of Fisher's exact test and odds ratios when testing $HRQ8_0$ are significant. The p-value is less then 0.05 and the odds ratio for fault-prone classes co-changing with anti-patterns is two and half times higher than for fault-prone other classes in the three programs.

We can answer positively to **RQ8**: we showed that classes co-changing with anti-patterns are significantly more fault-prone than other classes.

Indeed, in the three programs, we detect co-change situations with the majority of anti-patterns. In ArgoUML, Blob, LongMethod, and RefusedParentBequest co-change with other classes more than the rest of anti-patterns. During the evolution of JFreeChart and XercesJ, Blob is the anti-pattern that co-change the most with other classes.

But: We observe in Table 5.4.2, in the three analysed programs, that if a class belongs to the SpaghettiCode anti-pattern, it does not co-change with any other

	Faulty	Clean
Classes co-changing with AP in ArgoUML	241	102
Other classes in ArgoUML	1502	1480
Classes co-changing with AP in JFreeChart	68	26
Other classes in JFreeChart	674	847
Classes co-changing with AP in XercesJ	37	21
Other classes in in XercesJ	534	599
Total of classes co-changing with AP	346	149
Total of other classes	2710	2926
The p-value of Fisher's test	$2.2e-16$	
Odd-ratio	2.50723	

Table 5.3 – Contingency table and Fisher test results in ArgoUML, JFreeChart and XercesJ for classes with at least one fault (AP: Anti-patterns).

class in the system. In ArgoUML, we detect some occurrences of ClassDataShouldBePrivate, ComplexClass, and LongParameterList that co-changed with other classes. However, we do not detect any class playing role in these anti-patterns and that is co-changing with other classes in JFreeChart and XercesJ. We do not detect, also, classes that are co-changing with LongMethod classes in XercesJ. Finally, we found that classes that are co-changing with anti-patterns classes are significantly more fault prone than other classes with the same complexity, change history, and code size. However, it is impossible to exclude the possibility that there is no impact on fault-proneness for classes co-changed with SpaghettiCode, ClassDataShouldBePrivate, ComplexClass, and LongParameterList.

Other observations: Knowing that anti-pattern classes follow a change pattern implies the existence of (hidden) dependencies between the classes of this anti-pattern and other classes in the system. If these dependencies are not properly maintained, they lead to faults in the system. For example, the class `GoClassToNavigableClass.java` belong to a Blob anti-pattern in ArgoUML0.26. Concurrently, this class is co-changed with the class `GoClassToAssociatedClass.java`. Thus, these two classes must always be maintained together. Yet, in the Bugzilla of ArgoUML, the bug ID5505[7] confirms that the two classes are related but were not maintained together, leading to a fault.

[7] `http://argouml.tigris.org/issues/show_bug.cgi?id=5505`

Anti-patterns	Programs	# of CC	# of S.R.
AntiSingleton	ArgoUml	13	152
	JFreeChart	20	201
	XercesJ	18	188
Blob	ArgoUml	51	304
	JFreeChart	36	164
	XercesJ	24	93
ClassDataShouldBePrivate	ArgoUml	4	167
	JFreeChart	0	82
	XercesJ	0	113
ComplexClass	ArgoUml	2	192
	JFreeChart	0	146
	XercesJ	0	96
LongMethod	ArgoUml	42	282
	JFreeChart	51	314
	XercesJ	0	266
LongParameterList	ArgoUml	12	344
	JFreeChart	0	276
	XercesJ	0	309
MessageChains	ArgoUml	48	244
	JFreeChart	8	196
	XercesJ	16	183
RefusedParentBequest	ArgoUml	47	326
	JFreeChart	6	183
	XercesJ	25	93
SpaghettiCode	ArgoUml	0	0
	JFreeChart	0	0
	XercesJ	0	0
SpeculativeGenerality	ArgoUml	13	128
	JFreeChart	4	139
	XercesJ	8	201
SwissArmyKnife	ArgoUml	20	69
	JFreeChart	9	142
	XercesJ	18	108

Table 5.4 – Proportion of the anti-patterns dependencies (CC: co-changing situations of anti-patterns with other classes; S.R.: Anti-patterns static relationships).

5.5 Discussions

This section discusses the results reported in Section 5.4 as well as the threats to their validity.

5.5.1 Exploratory Findings

From Table 5.4.2, we note that many anti-patterns in ArgoUML, JFreechart, and XercesJ have static relationships and–or have been co-changed with other classes. To the best of our knowledge, we are the first to report these dependencies and to analyse their impact on fault proneness.

We do not consider that an anti-pattern is necessarily the result of a "bad" implementation or design choice; only the concerned developers can make such a judgement. We do not exclude that, in a particular context, an anti-pattern can be the best way to actually implement and–or design a (part of a) class. For example, automatically-generated parsers are often very large and complex classes. Only developers can evaluate their impact according to the context: it can be perfectly sensible to have these large and complex classes if they come from a well-defined grammar.

From Table 5.4.2, we report that different anti-patterns have different proportion of static relationships with other classes in programs. This difference is not surprising because these programs have been developed in three unrelated contexts, under different processes. It highlights the interest of analysing and reporting the anti-patterns dependencies when assessing finely the quality of programs.

SpaghettiCodes do not co-change and have no static relationships (use, association, aggregation, and composition) with other classes in the three analysed programs. First of all, we noted that we detect less SpaghettiCode occurrences than other anti-patterns occurrences in the different analyzed systems. Then, the nonexistence of SpaghettiCodes co-change and static relationshipsis is not surprising because a SpaghettiCode is revealed by classes with no structure, declaring long methods with no parameters and use global variables for processing. A SpaghettiCode does not exploit and prevents the use of object-orientation mechanisms: polymorphism and inheritance. With a SpaghettiCode, minimal relationships exist between objects. Many object methods have no parameters, and utilise classes or global variables for processing. Thus, a SpaghettiCode is difficult to reuse and to maintain, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse.

We found that classes that have dependencies with numerous anti-patterns (such as Blob and ComplexClass) are significantly more fault prone than other classes with the same complexity, change history, and code size. However, it is impossible to get significant statistical difference on fault proneness for some anti-patterns such as SpaghettiCode.

We also observe that many anti-patterns dependencies were with other motifs in programs such as design patterns. Design patterns are recurring solutions to common software design problems [32]. We noted that developers use design patterns, possibly unintentionally, as proven solutions to recurring design problems [46], *e.g.*, when there is a proliferation of similar methods and–or the user-interface code becomes difficult to maintain.

Last but not least, we confirmed that knowing that two classes are co-changing implies the existence of (hidden) dependencies between these two classes. If these dependencies are not properly maintained, they can introduce faults in a program [117]. We found that classes that co-changed with anti-patterns are more fault-prone than other co-changed classes in ArgoUML, JFreechart, and XercesJ. Thus, by knowing the sets of classes that co-changed with anti-patterns, we could explain and possibly prevent faults, thus lessening the anti-patterns negative impact. Indeed, team managers can guide programmers based on the program history and point out risky item coupling such as classes that are co-changing with anti-patterns classes. In addition, with the availability of such information, a tester could decide to focus on classes having dependencies with anti-patterns, because she knows that such classes are likely to contain faults.

5.5.2 Threats to Validity

We now discuss in details the threats to the validity of our results, following the guidelines provided in [110].

Construct validity. Threats concern the relation between theory and observation. In our context, they are mainly due to errors introduced in measurements. We are aware that the detection technique used includes some subjective understanding of the definitions of the anti-patterns. However, as discussed, we are interested to relate anti-patterns *as they are defined in DECOR* [96] with other

classes by static relationships *as they are defined in PADL* [35]. For this reason, the precision of the anti-patterns detection is a concern that we agree to accept. Moha *et al.* [96] reported that the DECOR current detection algorithms for anti-patterns ensure 100% recall and have a precision greater than 31% in the worst case, with an average precision greater than 60%. Macocha approach detection for macro co-change ensures 96% recall and have a precision greater than 85% [47]. While, for the detection of relationships among classes, the used approach ensures 100% recall and precision. We preprocessed the inconsistent anti-patterns to eliminate false positives. However, this preprocessing reduces the chances that we could answer our research questions wrongly. In addition, our results can still be affected by the presence of false negatives, *i.e.*, by a low recall exhibited by the anti-pattern detection tool. Nevertheless, in case that the anti-pattern specifications are variants of the specification used in DECOR, some anti-patterns may be missed during the detection phase. Although the sample of detected anti-patterns can be considered large enough to claim our conclusions, further investigations aimed at assessing to what extent the detection tool performance assess our results are needed.

Conclusion validity. These threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the statistical test that we used, Fisher's exact test, which is a non-parametric test. A possible threat to the conclusion validity is our particular choice for complexity and change metrics as representatives for the defect information contained in source code respective its change history. Although those metrics are widely used and accepted by other researchers there is no consensus as concerns their universality. We do not yet understand the complex mechanism of why and how defects are generated during the software development process. Thus, in theory there could be other, much more complex metrics hidden in source code, which are very powerful defect indicators but nobody discovered them yet.

Reliability validity. These threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to replicate our study. Moreover, both ArgoUML, JFreeChart, and XercesJ source code repositories publicly available, as well as the anti-pattern detection tool; the way our analysis

were performed is described in detail in Section 5.2. Finally, the data sets on which we computed our statistics are available on the Web[8].

Threats to **external validity.** These threats concern the possibility to generalise our observations. First, although we performed our study on three different, real programs belonging to different domains and with different sizes and histories, we cannot assert that our results and observations are generalisable to any other programs and the facts that all the analysed programs are in Java and open-source may reduce this generability. Nevertheless, it would be desirable to analyze further programs, also developed in different programming languages, to draw more general conclusions. Future work includes replicating our study with other programs. Second, we used particular, yet representative, sets of anti-patterns. Different anti-patterns could have lead to different results, which are part of our future work. In addition, the list of metrics used in our study is by no means complete. Therefore, using other metrics may yield different results. However, we believe that the same approach can be applied on any list of metrics. The odds ratio and p-value thresholds used in our study were chosen because they proved to be successful in previous studies [55].

5.6 Summary and Lessons Learned

A large amount of effort has been put into analysis models to explain and forecast faults in software programs. As this area of research grows, a greater number of metrics is being used to predict faults. In this chapter, we reported the results of an empirical study, performed on three object-oriented programs, which provides empirical evidence of the negative impact of dependencies with anti-patterns on fault-proneness. Through our two research questions:

- **RQ7:** RQ7: Are classes that have static relationships with anti-patterns more fault-prone than other classes?
- **RQ8:** RQ8: Are classes that co-change with anti-patterns more fault-prone than other classes?

we found that:

[8] `http://www.ptidej.net/download/experiments/msr12/`

- Having static relationships with anti-patterns can significantly increase fault-proneness.
- Classes having co-change dependencies with anti-patterns are more fault prone than others.

This empirical study confirms, within the limits of the threats to its validity, the conjecture in the literature that anti-patterns have a negative impact on system architecture. It also suggests to use the knowledge about the anti-patterns dependencies to maintain a system correctly, to eliminate design defects, and to propagate changes adequately.

Future work includes (1) replicating our study on other programs to assess the generality of our results, (2) studying the effect of the anti-patterns dependencies on change-proneness, and (3) analysing further the relationships among anti-patterns and design patterns not only through static analysis but also through co-change analysis.

CHAPTER 6

CONCLUSION

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

6.1 Summary

Over the years, many researchers have studied the evolution and maintenance of object-oriented source code to understand the costly decay of the programs in general and of their designs in particular.

Faults can occur when changes are made to a program by developers who do not and cannot fully understand artefacts dependencies among its artefacts and the original design [78]. Thus, in this thesis, we presented a set of novel approaches to improve the detection of co-change and co-evolution among program artefacts and to analyse the impact of the artefacts dependencies in terms of fault-proneness. We introduced the Asynchrony change pattern and the Dephase change pattern, as well as their approximate versions, to explain real scenarios of co-change and change propagation, which could help developers to maintain a program artefacts appropriately. We proposed an approach, Macocha, which mines software repositories and uses several algorithms and techniques, such as the k-nearest neighbor algorithm, the Hamming distance, and a bit vector model, to discover occurrences of the (approximate) Asynchrony and Dephase change patterns.

Macocha relates to file stability and co-changes. We therefore performed two types of empirical studies. Quantitatively, we compared Macocha with UMLDiff [106] and an association rules-based approach [115] by applying and comparing the results of the three approaches on seven programs: ArgoUML, FreeBSD, JFreeChart, Openser, SIP, XalanC, and XercesC. We showed that Macocha has better precision and recall than the state-of-the-art approaches based on association rules [115]. Qualitatively, we used external information and static analysis to show that detected MCCs and DMCCs explain real, important evolution phe-

nomena. We also showed that occurrences of Dephase change patterns exist and help in explaining bugs, managing development teams, and performing traceability analysis.

We use Macocha to analyse the impact of the evolution of object-oriented programs and to precise the impact of design defect on fault-proneness. Therefore, we proposed a second approach, Profilo, to study the evolution of classes to detect the impact of this evolution on fault-proneness. We mined source code and version control programs to create class-profiles for classes and detect similar evolution class-profiles, which represent a co-evolution profile, *i.e.*, two or more classes that evolve together and whose structures have been modified in the same versions over the whole lifespan of the program. By applying Profilo on three Java programs, we found that faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes. We also showed that classes that have not been removed since their first appearance in a given version of a program, *i.e.*, Persistent classes, are significantly less fault-prone than other classes.

Finally, we performed a series of experiments aimed at understanding and assessing the impact of anti-patterns dependencies on classes fault-proneness. From these experiments, we drew the following conclusions: classes having static relationships or that have been co-changed with anti-patterns are significantly more likely to be involved in fault-fixing commits than other classes. Thus, developers should pay attention to programs with a high number of such classes, because they are more likely to be the subject of their maintenance efforts. Thus, we answer our thesis by drawing the following conclusions and contributions:

1. The ability to analyse file stability for any program, providing that their CVS/SVN repositories are available.

2. The detection of several occurrences of the (approximate) Asynchrony and Dephase change patterns (two novel change patterns) in different programs belonging to different domains and with different sizes, histories, and programming languages.

3. The scope of macro co-change could improve the identified co-changes over an approach based on association rules in terms of precision and recall.

4. The combination of structural and textual similarities for class renamings detection provides the possibility to analyse the evolution of classes among different versions of a program.

5. The bit-vector algorithm, which was effectively adapted to class diagram matching, gives valuable insight about a program evolution.

6. Persistent classes are less fault-prone than Short-lived, and Transient classes.

7. Faults fixed by maintaining co-evolved classes are more frequent than faults fixed using not co-evolved classes.

8. Anti-patterns have, through their dependencies, a negative impact on program architecture in term of fault-proneness.

9. The detection of design defects dependencies in general: although we train our approach on only eleven kinds of design defects, it can detect any static relationship and co-change dependency for any kind of design defects.

6.2 Opportunities

We showed our thesis by considering the presence external information, statistics analysis, and some metrics such as precision and recall. This thesis opens new directions of research including: using change patterns to predict changes, identifying risky parts of programmes and "buggy" changes, and the extension of different approaches presented in this thesis as follow:

6.2.1 Using Change Patterns to Predict Changes

Estimation of the change-proneness of different artefacts is an active research topic in the area of software engineering. Such estimation can be used to predict changes to different artefacts of a program from one release to the next. It can also be used to estimate and possibly reduce the effort required during the development and maintenance phase by balancing the amount of developers time assigned to each part of a program.

We could adapt Asynchrony and Dephase change patterns to predict changes to classes. We will study the possibility of using the data about previous changes to forecast future changes that could occur in a program, their potential damages, and the factors influencing their propagations.

We are currently (1) relating Asynchrony and Dephase change patterns to program quality and external software characteristics, such as change proneness(2) identifying other scenarios in which Asynchrony and Dephase change patterns help in reducing maintenance costs, and (3) evaluating the consistency and usefulness of change patterns occurrences, including files recently changed.

6.2.2 Identifying Risky Parts of Programmes and "Buggy" Changes

In this thesis, we highlighted that software evolution is related with fault-proneness. Therefore, we could propose to improve the maintenance of a program by identifying refactoring opportunities, which resolve design defects existing in source code. Thus, we could provide solutions to help developers in improving design quality by appropriate refactorings.

Change classification learns from the change history of a program to classify any future change as clean or buggy. Indeed, changes can be classified as either buggy or clean by using a trained classifier model. In this manner, the change classification predicts whether a new change is more similar to prior "buggy" or clean changes.

However, very few studies investigated the use of knowledge deducted from co-change dependencies to classify changes as "buggy" or clean. It would be interesting to understand how co-changes dependencies affect developers' behaviors as well as their ability to write good code. We could provide feedback to developers on the quality of their code each time a new co-change pattern is detected through series of experiments assessing the usability of a quality model using change classification and co-change dependencies.

6.2.3 Extending the Approaches Presented in this Thesis

Although we have proved in this thesis that dependencies among artefacts and motifs, such as change patterns and anti-patterns, significantly impact fault-proneness, it would be interesting to assess their impact on more subjective quality attributes like understandability. We could perform a series of controlled experiments to understand the effect of various anti-patterns and change patterns on the understandability of programs. In the near future, we could I plan to investigate new sources of information in software repositories, such as mailing lists.

Indeed, a mailing list allows for widespread distribution of information to many developers. Recent studies [91] have found that mailing list activity is closely related to source code activity and mailing list discussions are good indicators of the types of source code changes being carried out on a program. It would be interesting to investigate whether a combination of software evolution analysis and mailing lists mining improves fault proneness detection.

PUBLICATIONS

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

6.2.3.0.1 Articles in journal

1. **Fehmi Jaafar**, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. Detecting Asynchrony and Dephase Change Patterns by Mining Software Repositories. Journal of Software Maintenance and Evolution: Research and Practice (accepted).

2. **Fehmi Jaafar**, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. Analysing Anti-patterns Static Relationships with Design Patterns. Journal of Electronic Communications of the European Association of Software Science and Technology (under revision).

6.2.3.0.2 Conference articles

1. **Fehmi Jaafar**, Yann-Gaël Guéhéneuc, Sylvie Hamel , and Foutse Khomh. Mining the Relationship Between Anti-patterns Dependencies and Fault-proneness. 20th edition of the Working Conference on Reverse Engineering. October 14-17, 2013, Koblenz-Landau, Germany.

2. Nasir Ali, Ahmed E. Hassan, and **Fehmi Jaafar**. Leveraging Historical Co-Change Information for Requirements Traceability. 20th edition of the Working Conference on Reverse Engineering. October 14-17, 2013, Koblenz-Landau, Germany.

3. **Fehmi Jaafar**, Salima Hassaine, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Bram Adams. On the Relationship Between Program Evolution and Fault-proneness: An Empirical Study. 17th European Conference on Software Maintenance and Reengineering March 58, 2013, Genova, Italy.

4. **Fehmi Jaafar**, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Analysing Anti-patterns Static Relationships with Design Patterns. First Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP). March 2013, Italy.

5. **Fehmi Jaafar** On the analysis of evolution of software artefacts and programs. 34th International Conference on Software Engineering (ICSE Ph.D. symposium). June 2012, Switzerland.

6. **Fehmi Jaafar**, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Sylvie Hamel. An Exploratory Study of Macro Co-changes. 18th Working Conference on

Reverse Engineering (WCRE). October 2011, Ireland. (Invited to the Journal of Software Maintenance and Evolution: Research and Practice)

BIBLIOGRAPHY

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[3] Nasir Ali, Yann-Gaël Gueheneuc, and Giuliano Antoniol. Trust-based requirements traceability. In *Proceedings of the IEEE 19th International Conference on Program Comprehension*, pages 111–120. IEEE Computer Society, 2011.

[4] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. *Principles of Software Evolution, International Workshop on*, pages 31–40, 2004.

[5] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi. Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories. In *Proceedings of the International Workshop on Mining software repositories*, pages 1–5. ACM Press, 2005.

[6] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *Foundations of Software Engineering*, pages 385–394, New York, NY, USA, 2007. ACM Press.

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

[8] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *Software*, pages 728–738, 1984.

[9] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 125–134. ACM, 2010.

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

[11] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth International Workshop on Principles of Software Evolution*, pages 11–18. ACM, 2007.

[12] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 199–210. IEEE Computer Society Press, 2006.

[13] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. Dependence anti patterns. In 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability*, pages 25–34, 2008.

[14] 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.

[15] Salah Bouktif, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Salah Bouktif, Yann-Gael Gueheneuc, and Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 221–230, Los Alamitos CA USA, 2006. IEEE Computer Society. ISBN 1095-1350.

[17] W.J. Brown, H.W. McCormick, T.J. Mowbray, and R.C. Malveau. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[18] 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 29–38. IEEE Computer Society Press, 2005.

[19] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society Press.

[20] Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 163–166. ACM Press, 2010.

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

[22] Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *Transactions on Software Engineering*, pages 720–735, 2009.

[23] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 135–144. IEEE Computer Society, 2009.

[24] B.V. Dasarathy. *Nearest Neighbor (NN) Norms:NN Pattern Classification Techniques*. IEEE Computer Society Press, 1991.

[25] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM.

[26] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transaction Software Engineering*, pages 1–12, 2001.

[27] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–33. IEEE Computer Society, 2003.

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

[29] William B. Frakes and Ricardo A. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[30] Christopher Fraser. Difftree: Inferring phylogenies for evolving software. Technical report, Microsoft Research, 2005.

[31] 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*, pages 190–200. IEEE Computer Society, 1998.

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

[33] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, pages 369–393, 2006.

[34] 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* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–314. ACM Press, October 2004.

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

[36] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Khashayar Khosravi, and Houari Sahraoui. Design patterns as laws of quality. In *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*, pages 1–35. University of Montreal, 2005.

[38] Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. Measuring class importance in the context of design evolution. In *Proceedings of the IEEE 18th International Conference on Program Comprehension*, pages 148–151. IEEE Computer Society, 2010.

[39] Richard Hamming. Error detecting and error correcting codes. *BELL system technical journal*, pages 147–160, 1950. ISSN 0005-8580.

[40] 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.

[41] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

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

[43] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272. IEEE Computer Society, 2005.

[44] Les Hatton. How accurately do engineers predict software maintenance tasks? *Computer*, pages 64–69, 2007.

[45] Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration: a survey. *Information Software Technology*, pages 643–656, 2005.

[46] Claudia Iacob. A design pattern mining method for interaction design. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 217–222. ACM, 2011.

[47] Fehmi Jaafar, Guéhéneuc Yann-Gaël, Sylvie Hamel, and Antoniol Antoniol. An exploratory study of macro co-changes. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 325–334. ACM, 2011.

[48] Fehmi Jaafar, Salima Hassaine, Yann-Gael Gueheneuc, Sylvie Hamel, and Bram Adams. On the relationship between program evolution and fault-proneness: An empirical study. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pages 15–24, Washington, DC, USA, 2013. IEEE Computer Society.

[49] Fehmi Jaafar, Guéhéneuc Yann-Gaël, Sylvie Hamel, and Foutse Khomh. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. ACM, 2013.

[50] Huzefa Kagdi and Jonathan I. Maletic. Software repositories: A source for traceability links. In *in Proceedings of 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 22–23. IEEE Computer Society, 2007.

[51] Huzefa Kagdi, Jonathan I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 145–154. IEEE Computer Society, 2007.

[52] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, pages 13–23, 2005.

[53] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 351–360. ACM, 2011.

[54] 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 Softw. Engg.*, 17(3):243–275, 2012.

[55] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Guiliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, pages 243–275, 2012.

[56] Doug Kimelman, Marsha Kimelman, David Mandelin, and Daniel M. Yellin. Bayesian approaches to matching architectural diagrams. *Software Engineering, IEEE Transactions on*, pages 248–274, 2010.

[57] SB Kotsiantis, ID Zaharakis, and PE Pintelas. Supervised machine learning: A review of classification techniques. *Frontiers in artificiel intelligence and applications*, pages 3–24, 2007.

[58] 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 *European Conference on Software Maintenance and Reengineering*, pages 179–188, 2009.

[59] 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.

[60] 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*, pages 139–163, 2013.

[61] D. L. Lanning and T. M. Khoshgoftar. Canonical modelling of software complexity and fault correction activity. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 374–381, Victoria, 1994.

[62] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Languages and Models with Objets*, pages 135–149. Lavoisier, 2002.

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

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

[65] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, 1996.

[66] M. M. Lehman. Feedback in the software evolution process. *Information and Software Technology*, pages 681–686, November 1996.

[67] M. M. Lehman and L. A. Belady. *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

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

[69] 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.

[70] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *Software Engineering, IEEE Transactions on*, pages 287–300, 2008.

[71] Radu Marinescu and Cristina Marinescu. Are the clients of flawed classes (also) defect prone? In *Proceedings of the IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 65–74, Washington, DC, USA, 2011. IEEE Computer Society.

[72] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. on Software Engineering and Methodology*, 11(3):309–346, 2002.

[73] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, New York, NY, USA, 2008. ACM.

[74] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[75] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[76] 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.

[77] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transaction on Software Engineering*, pages 340–355, 2005.

[78] 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.

[79] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, pages 40–52, 1992.

[80] B. Pietrzak and B. Walter. Leveraging code smell detection with inter-smell relations. *Extreme Programming and Agile Processes in Software Engineering*, pages 75–84, 2006.

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

[82] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*, pages 223–233. IEEE Computer Society, 2004.

[83] Brian Wilkerson Rebecca Wirfs-Brock and Lauren Wiener, editors. *Designing Object-Oriented Software*. Academic Press Professional, Inc., 1990.

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

[85] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[86] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source

code changes. *Working Conference on Reverse Engineering*, pages 437–446, 2012.

[87] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[88] Dimitrios Settas, Antonio Cerone, and Stefan Fenz. Enhancing ontology-based antipattern detection using bayesian networks. *Expert Systems with Applications*, pages 9041–9053, 2012.

[89] SyedMuhammadAli Shah and Maurizio Morisio. Complexity metrics significance for defects: An empirical view. In *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, pages 29–37. Springer Berlin Heidelberg, 2013.

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

[91] Emad Shihab, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. On the central role of mailing lists in open source projects: an exploratory study. In *Proceedings of the international conference on New frontiers in artificial intelligence*, pages 91–103, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] Gagandeep Singh and Hardeep Singh. Effect of software evolution on metrics and applicability of lehman's laws of software evolution. *SIGSOFT Software Enggineering Notes*, pages 1–7, 2013.

[93] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories MSR 2005 Saint Louis Missouri USA*, May 17 2005.

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

[95] Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Playing roles in design patterns: An empirical descriptive and analytic study. In Kostas Kontogiannis and Tao Xie, editors, *Proceedings of the* 25^{th} *International Conference on Software Maintenance (ICSM)*, pages 83–92. IEEE Computer Society Press, September 2009.

[96] 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)*, pages 1–17 pages, 2009.

[97] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society.

[98] 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. *J. Software Maintenance and Evolution*, pages 277–306, 2005.

[99] Adam Vanya, Steven Klusener, Nico van Rooijen, and Hans van Vliet. Characterizing evolutionary clusters. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 227–236. IEEE Computer Society, 2009.

[100] Adam Vanya, Rahul Premraj, and Hans van Vliet. Approximating change sets at philips healthcare: A case study. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 121–130. IEEE Computer Society, 2011.

[101] Marek Vokavc. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Software Eng.*, 30:904–917, 2004.

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

[103] 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.

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

[105] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *16th Internatioanl Conferance on Software Engineering and Knowledge Engineering*, pages 123–128. Citeseer, 2004.

[106] Zhenchang Xing and Eleni Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *Transactions on Software Engineering*, 31:850–868, 2005.

[107] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering*, pages 54–65. ACM Press, 2005.

[108] Zhenchang Xing and Eleni Stroulia. Bottom-up design evolution concern discovery and analysis. Technical report, University of Alberta, 2007.

[109] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *International Conference on Software Maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[110] Robert K. Yin. *Case Study Research: Design and Methods - Third Edition.* SAGE Publications, London, 2002.

[111] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, pages 574–586, 2004.

[112] He (Jason) Zhang, Juan Li, Liming Zhu, Ross Jeffery, Jenny Liu, and Qing Wang. Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology*, pages 1–18, 2013.

[113] 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*, pages 27–36. IEEE Computer Society, 2008.

[114] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, pages 531–540. ACM, 2008.

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

[116] Thomas Zimmermann, Silvia Breu, Christian Lindig, and Benjamin Livshits. Mining additions of method calls in argouml. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 169–170. ACM Press, 2006.

[117] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 9–16, Washington, DC, USA, 2007. IEEE Computer Society.

APPENDIX A

DEFINITIONS OF METRICS AND QUALITY ATTRIBUTES

This Appendix presents the definitions of the quality attributes [7, 32, 37], and all the metrics used in this dissertation.

A.1 Metrics

Total Lines of Code (TLOC): Measures the total number lines of code of class.

Fan out (FOUT): Measures the number of method calls of a class.

Nested Block Depth (NBD): Measures the nested block depth of the methods in a class.

Number of Parameters (PAR): Measures the number of parameters of the methods in class.

McCabe Cyclomatic Complexity (VG): Measures the McCabe cyclomatic complexity of the methods in a class.

Number of Fields (NOF): Measures the number of fields of the classes.

Number of Methods (NOM): Measures the number of methods of the classes.

Total Prior Changes (TPC): Measures the total number of changes to a class in the 6 months before the release.

Prior Bug Fixing Changes (BFC): The number of bug fixing changes done to a class in the 6 months before the release.

Pre-release faults (PRE): The number of pre-release faults in a class in the 6 months before the release.

Post-release defects (POST): The number of post-release defects in a class in the 6 months after the release.

A.2 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

DEFINITIONS OF CODE SMELLS AND ANTI-PATTERNS

This Appendix presents the definitions of code smells and anti-patterns studied in this dissertation.

B.1 Code Smells

In this dissertation we focused on the following code smells:

AbstractClass: this code smell is characteristic of the Speculative Generality Anti-pattern. 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 Anti-patterns

This dissertation focused on the following anti-patterns:

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 [84]) 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 [104].

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 [28].

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 anti-pattern 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.