POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Linguistic anti-patterns: Impact analysis on code quality

ZEINAB KERMANSARAVI
Département de génie informatique et génie logiciel

Thése présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique
Juillet 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée:

Linguistic anti-patterns: Impact analysis on code quality

présentée par **Zeinab KERMANSARAVI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de:

Giovanni BELTRAME, présidente
Foutse KHOMH, membre et directeur de recherche
Yann-Gaël GUÉHÉNEUC, membre et codirectrice de recherche
Jinghui CHENG, membre
Eugene SYRIANI, membre externe

DEDICATION

This dissertation is dedicated to
To Hani and Anita
To my parents
To my family
For their endless love, support and encouragement.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisors, Foutse and Yann, for encouraging and believing in me. For everything I learned from them, for all the opportunities they gave me, for their guidance with their extensive knowledge when I was lost, for their patience when I needed them to listen, for being honest when they had doubts about ideas and methodologies, for being not so honest when answering "I don't know" because they wanted me to figure it out.

I would also like to thank the members of my Ph.D. committee who enthusiastically accepted to review my dissertation. I also thank all friends and colleagues, current and past members of the Ptidej, SWAT, Soccerlab and MCIS teams, for sharing ideas, providing feedback and the productive discussions.

I am very thankful to my whole family : To my mother who somehow convinced me that 'failure' is not a dictionary word ; to my father to whom I never managed to explain what I do but who is always happy for my success ; to my brother who was always there for me. To my brothers. I love you all so much!

Last but certainly not least, I thank my beloved husband Hani for letting me pursuing my dreams. Thank for your unconditional support, for your patience, and understanding. Thank you for postponing your plans to catch up with mines. Thank you for loving me!

I also thank my little angel (my daughter Anita) who was born on June 13th 2016 (during my Ph.D.) and gave me an infinite source of energy by her endless smiles. She brought me all the luck by opening her eyes to this beautiful word.

RÉSUMÉ

Les “mauvaises odeurs” de conception sont des structures qui indiquent une violation des principes fondamentaux de conception et qui nuisent à la qualité des systèmes logiciels. Ils représentent des choix d’architectures, de conception, et d’implémentation qui doivent être suivis et améliorés. Dans ce travail, on considère deux sous types de ces “mauvaises odeurs” qui sont les anti-patrons de conception (DAPs) et les anti-patrons linguistiques (LAs).

Les anti-patrons de conception (DAPs) sont les patrons que les développeurs considèrent comme étant des bonnes solutions à certains problèmes mais qui ont en réalité un impact négatif sur la qualité des logiciels. Des études récentes ont démontré que les anti-patrons rendent la maintenance logicielle plus difficile dans les systèmes orientés objets ainsi qu’ils augmentent le changement et les défaillances.

Le concept d’anti-patrons linguistiques (LAs) fait référence aux mauvaises pratiques de nommage, de documentation et de l’implémentation du code source qui peuvent négativement impacter la qualité des systèmes logiciels et la compréhension du programme.

Contrairement aux anti-patrons, les patrons de conception (DPs) présentent une solution prometteuse qui sert à améliorer la qualité des systèmes orientés objets. Dans certains cas, les patrons de conception et contrairement à ce qui est connu, peuvent avoir aussi un impact négatif sur la qualité logicielle. Pour cela, nous considérons également les patrons de conception dans ce travail afin d’étudier leurs comportements et leurs qualité au cours de l’évolution des logiciels.

Avoir une bonne qualité logicielle est primordial pour contrôler et réduire les coûts de la maintenance des systèmes orientés objets. Il est important de disposer de mécanismes permettant de mesurer la qualité logicielle. Cependant, la qualité a des différentes significations qui peuvent être par exemple la capacité d’un système à changer à faible coût ou même l’absence de bogues dans le logiciel. Dans cette thèse, nous considérons comme mesures indirectes de la qualité: la compréhension du code, la propension au changement, et la prévalance de fautes.

Au cours de l’évolution d’un logiciel, les développeurs risquent d’introduire des anti-patrons durant leurs tâches de développement (fixer des bogues, ajouter des nouvelles fonctionnalités, ou même appliquer des nouvelles exigences). Dans cette thèse, nous avons étudié l’impact des anti-patrons de conception, les anti-patrons linguistiques, et les patrons de conception sur la qualité logicielle.

Premièrement, nous avons réalisé une étude empirique afin d’investiguer si les occurrences des LAs impactent réellement la compréhension du code. Nous avons remarqué que les LAs affectent négativement la compréhension; en diminuant le nombre des réponses correctes lors des exercices

de maintenance. Egalement, nous avons étudié le rôle des connaissances des LAs sur l'effet des leurs occurrences et nous avons constaté que ces connaissances peuvent atténuer l'impact négatif de ces LAs.

Deuxièmement, nous avons utilisé la propension au changement et aux fautes pour capturer l'impact des LAs et des anti-patrons sur la qualité. Nous avons commencé par étudier la relation entre LAs et la qualité ainsi que leur interaction avec les anti-patrons. Nous avons constaté que les classes qui contiennent uniquement des anti-patrons sont plus susceptibles de changer et sont plus susceptibles de subir des erreurs de code que les classes qui contiennent des LAs uniquement.

Troisièmement, nous avons investigué l'évolution des LAs, des anti-patrons, et des patrons de conceptions. Nous avons constaté qu'ils se transforment d'un type d'anti-patron à un autre. Ils se transforment en patron de conceptions lorsqu'ils sont corrigés. Aussi, les patrons de conceptions peuvent se dégrader en anti-patrons.

Nous présentons également les types de changements qui déclenchent les LAs, les anti-patrons, et les mutations des patrons de conception. Ainsi, nous apportons des preuves concrètes de l'impact des mauvaises odeurs de conception sur la qualité logiciel.

ABSTRACT

Design smells are bad practices in software design that lower the quality of software systems. They represent architectural, design, and implementation choices that should be tracked and removed. We consider design anti-patterns (DAPs) and linguistic anti-patterns (LAs) as two special types of design smells in our work, in contrast to design patterns (DPs).

DAPs are software patterns that are thought by developers to be good solutions to some design problems but that have actually a negative impact on quality. Recent studies have brought evidence that DAPs make maintenance more difficult in object-oriented systems and increase change- and fault-proneness.

LAs refer to bad practices of naming, documentation, and implementation of code entities, which could decrease the quality of software systems and have a negative impact on program comprehension.

Opposite to design smells, DPs are promising solutions to improve the quality of object-oriented systems. Yet against popular wisdom, design patterns in practice can impact quality negatively.

Achieving good quality is important to control and reduce the maintenance cost of object-oriented systems. This goal requires means to measure the quality of systems. However, quality has different meanings, e.g., the capacity of a system to change at low cost or the absence of bugs. In this thesis, we consider code understanding, change-proneness, and fault-proneness as three proxy measures for quality.

During software evolution, which is a never-ending activity, developers may introduce DAPs and LAs, when they modify software systems to fix bugs or to add new functionalities based on changes in requirements. Developers may also use design patterns to ensure software quality or as a possible cure for some design smells. Thus, DPs, DAPs, and LAs are introduced, removed, and mutated from one to another by developers. In this thesis, we studied the mutations of LAs, DAPs, and DPs and their impact on quality. To achieve our goal, we conducted three different studies as follows:

First, we perform an empirical study to investigate whether the occurrences of LAs do impact code understandability. We observe that LAs negatively affect participants' understanding; decreasing the numbers of correct answers during code understandability exercises. We also study the role that prior knowledge of LAs has on the effect of LA occurrences on code understandability and observe that prior knowledge can mitigate the negative impact of LAs.

Second, we use change- and fault-proneness to capture the impact of LAs and DAPs on quality. We first study the relationship between LAs and quality as well as their interaction with DPAs. The

results show that classes that contain both DAPs and LAs are more change- and fault-prone than the other classes, which only have LAs or DAPs. We also find that classes containing only DAPs are more change- and fault-prone than classes with LAs only.

Third, we investigate the evolution of DAPs and DPs and find that they mutate into other DPs and–or DAPs. We also find that some change types (renaming, and changes in comments, declarations, and operators) primarily trigger mutations. In terms of fault-proneness, we find that DAPs are more fault-prone than DPs and mutation of DAPs to DPs is the most faulty type of mutation. Finally, we study whether the existence of LAs makes classes participating in evolving DPs or DAPs more change- and fault-prone. We find that classes containing DAPs and LAs are more change- and fault-prone than classes containing DPs and LAs, or only each type of smells individually.

Thus, we bring concrete evidence on the impact of these design smells on quality in terms of code understandability, change-, and fault-proneness. These results also provide important insights into the evolution of DPs and DAPs and its impacts on the change- and fault-proneness of software systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Research Context

Quality is one of the most important challenges during development and maintenance of software systems. A development team may implement software features with poor design, bad coding, naming and documentation issues, which are collectively called "design smells". We focus on linguistic anti-patterns (LAs) and design anti-patterns (DAPs). Design anti-patterns include relations among classes and linguistic anti-patterns pertain to the naming of identifiers used in the code and design.

Linguistic anti-patterns (LAs) are one type of smell, which refer to incompatibility in naming, documentation, and implementation of an entity. Arnaoudova *et al*. (2013) was the first to introduce the concept of linguistic anti-patterns (LAs), to provide a catalog of LAs, and to categorize them into "Method" and "Attributes" types of LAs.

Design anti-patterns (DAPs) are "poor" solutions to recurrent design problems that make object-oriented systems difficult to maintain.

In addition, all software systems must evolve frequently to fix problems, bugs, and even add some parts for new requirements. This evolution may decay their implementation and cause common bad practice solutions like design anti-patterns which have a potential negative impact on quality. On the other hand, developers may use design patterns (DPs) when changing their systems, either because the modification calls for some design patterns or as a possible cure for some anti-patterns.

1.2 Problem Statement

Hofmeister *et al*. (2017) showed that shorter identifier names impact program comprehension negatively. Besides, Arnaoudova *et al*. (2016) performed an experiment on LAs by examining the developers' perception of the quality of code snippets containing LAs and reported that developers consider LAs to be poor practices that should be refactored. Fakhoury *et al*. (2018) studied the effect of LAs and readability on developers' cognitive load using a minimally-invasive functional brain-imaging technique (functional Near Infrared Spectroscopy, fNIRS) and an eye-tracker. They reported that LAs have a negative impact on developers' cognitive load. Although these previous works established a relation between the occurrence of LAs and program comprehension, they did not investigate their impact on code quality in terms of change- and fault-proneness. Neither did they studied if different types of LAs affect comprehension differently. Therefore, we

study whether different types of LAs affect code understandability equally. We also study whether having knowledge of the LAs could have a positive effect on developers' understanding of code containing LAs.

In this thesis, we want to study the impact of the occurrences of LAs on the code quality in terms of change- and fault-proneness as well. Since, DAPs and DPs often co-exist in software systems and since previous studies report that they impact software quality, we consider them as a side factor, to investigate how the occurrences of LAs on the classes containing DAPs, DPs, or both could impact on the quality in terms of change- and fault-proneness.

Some of the previous works, (Khomh *et al*., 2012), (Taba *et al*., 2013), (Yamashita et Moonen, 2013) reported that the occurrence of design anti-patterns in systems increases change- and fault-proneness. Some of the other studies investigated the relationships between DAPs and DPs in releases of several systems independently, one release at a time (Jaafar *et al*., 2013, 2014).
Jaafar *et al*. (2013) studied the relationships between classes playing roles in DPs and DAPs. They found that there is a relationship between DPs and DAPs but they are temporary. Jaafar *et al.* also showed that classes included in DAPs which have such relationships with DPs are more change-prone than other DAP classes, but are less fault-prone than others. Since the concept of LAs is new (Arnaoudova *et al*., 2013) and there is no study that investigated both LAs and DAPs and their impact on quality, we were encouraged to study the relation between LAs and DAPs to see how their co-occurrence impacts change- and fault-proneness.

Finally, in a previous study (Khomh et Guéhéneuc, 2008) have shown that the design of systems degrade over time, presumably due to the removal (or lack of use) of DPs and the introduction of DAPs. Accordingly, understanding the dynamics behind the evolution of DPs and DAPs, in particular their mutations, could help development teams better prioritize maintenance activities and the allocation of resources. Therefore, we were encouraged to investigate such mutations, and analyze their impact on change- and fault- proneness. Although we are aware of the negative impacts of LAs on code quality and comprehension, it is important to know the impacts of the co-occurrence of LAs and DPs or DAPs on the quality of software systems, particularly on the change- and fault-proneness. As a further analysis from our second study, in Chapter 5, we also study the impact of the occurrences of LAs on the quality of software systems during their evolution.

1.3 Research Goal

The main goal of our research is to help and assist developers to understand, prevent, and correct smells (LAs, and DAPs), which may increase the risk of faults or failures in the future. We address our research goal through the following studies:

- Investigating the impact of different types of LAs on code understandability.
- Studying the effect of prior knowledge of LAs on developers' understanding of code containing LAs.
- Studying the impacts of LAs on software systems' change- and fault-proneness?
- Studying the probability of design anti-patterns and design pattern mutations using Markov models.
- Investigating how design patterns and–or design anti-patterns mutate over time by modeling the behavior of these mutations using Markov models.
- Examining the impact of these mutations on change- and fault-proneness.
- Studying the types of changes to the code that lead to design patterns and design anti-patterns mutations.
- Investigating the most fault-prone transitions from design patterns and–or design anti-patterns.
- Studying the impact of the occurrences of LAs on the change and fault-proneness of mutated classes.

1.4 Contributions

In this dissertation, we studied the impact of different smells and patterns on the quality of software systems. To reach our research goal, we did the following contributions:

1.4.1 Linguistic anti-patterns and Program Comprehension

We undertake an empirical study to find "How different factors affect the impact of LAs on the program comprehension?". We examine 7 types of LAs from 10 systems.

To answer this question, we perform two experiments ExpBefore and ExpAfter investigating whether seven different types of LAs affect code understandability equally. These two experiments allow us to also investigate whether knowing the characteristics of LAs improve developers' understandability of code snippets containing LAs. We use three criteria to evaluate the participants' performance, including (1) participants' effort using NASA TLX, (2) the time they spent to do the tasks using a hidden automatic timer, and (3) their percentages of correct answers (by comparing their answers to our oracle). The results show that LAs have a negative impact on understandability. We also

observed that, in general, having knowledge about LAs help participants finding the right answers more easily and faster.

Moreover, we collect some information from all participants such as their age, gender, work experience, programming experience, English level, and etc, in order to report which participants with which profile could find LAs, and answer the question better.

Besides, we investigate how these different factors affect the participants' performance in the same situation. We also found that work experience and LAs knowledge as well as level in English positively impact on the measured variables. Therefore, we study the impact of English skill on understandability by investigating the collected self-reporting data, and comparing English and French speakers' results. We found that a proficient English skill has a positive impact on program comprehension.

This information could be helpful for developers and companies to know which profiles match better with specific job positions. The details of the study have been provided in Chapter 4.

Our results are currently under the second review in the Software Quality Journal (SQJ):

- Zeinab (Azadeh) Kermansaravi, Diana El-Masri, Foutse Khomh, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-lhadj. *A Large-Scale Empirical Study of the Impact of Linguistic Anti-patterns on Program Comprehension*. Submitted to The Software Quality Journal (SQJ), April 2019, Under the second revision as of May, 24th, 2019.

1.4.2 Linguistic Anti-patterns, Design Anti-patterns and their impact on Change-, and Fault-Proneness

We conduct another empirical study to evaluate the relationship between LAs and software quality as well as their interaction with design anti-patterns. We investigate whether LAs can have an additional impact on change- and fault-proneness when occurring on classes with design anti-patterns. Specifically, we compare in terms of change- and fault-proneness between (1) classes with both DAPs and LAs and classes with DAPs only, (2) classes containing both DAPs and LAs and classes with LAs only, as well as (3) classes with DAPs only and those with LAs only.

We detect 29 design smells consisting of 13 DAPs and 17 LAs in 30 releases of three projects: ANT, ArgoUML, and Hibernate. We analyze to what extent classes containing LAs have higher (or lower) odds to change or to be subject to fault fixing than other classes containing DAPs. The results show and bring empirical evidence on the fact that LAs can make, in some cases, classes with DAPs more fault-prone.

In addition, we empirically demonstrate that classes containing only DAPs are more change- and

fault-prone than classes with LAs only.

We believe such results could guide development and quality assurance teams to better focus their refactoring efforts on components with design smells (without neglecting linguistic anti-patterns) to assure good quality for their systems. We have provided all details of this study in Chapter5.

Our results have been published in Software Quality Journal (SQJ):

- Latifa Guerrouj, Zeinab (Azadeh) Kermansaravi, Venera Arnaoudova, Benjamin C. M. Fung, Foutse Khomh, Giuliano Antoniol, Yann-Gaël Guéhéneuc. *Investigating the relation between lexical smells and change- and fault-proneness: an empirical study*. Software Quality Journal (SQJ), May 2016, Springer.

1.4.3 Linguistic anti-patterns, Design anti-patterns, Design patterns, their mutations and Change- and Fault-Proneness

We perform our last study on seven software systems; examining 13 design anti-patterns, 8 design patterns, and all types (17) of LAs. The results show that, indeed, DPs and DAPs mutate into other DPs and–or DAPs.

Besides, we observed that these mutations are not random and we build a Markov model to capture the probability of occurrences of the different mutations. Using this model; we show that these mutations affect the fault-proneness of the classes participating in the mutating DPs and DAPs differently.

We also identified the change types that trigger the mutations. At the end, we also investigate the role of LAs on classes contains DPs and DAPs and find classes containing both DAPs and LAs to be more change- and fault-prone than classes containing both DPs and LAs, as well as classes that only have DAPs or LAs or DPs.

Using this information, developers can focus on the design patterns that are most likely to mutate into design anti-patterns and–or to have more fault-proneness. Thus, this information can help development and quality assurance teams to better focus their refactoring efforts. This is likely to be useful to control and improve the quality of their systems. The results of this study are currently under review in the Empirical software engineering Journal (EMSE):

- Zeinab (Azadeh) Kermansaravi, Md. Saidur Rahman, Foutse Khomh, Yann-Gaël Guéhéneuc. *Investigating Design Patterns and Design Anti-pattern Mutations and Their Fault-proneness*. Empirical software engineering Journal (EMSE), June 2019.

Based on our three contributions and results, developers and companies (1) can focus on specific

types of LAs that have the most negative impact on the code understandability and refactor them to improve the code understandability; (2) can refactor the classes that contain DAPs, which participate in LAs, to prevent changes and faults; (3) avoid DAPs and DPs mutations that lead to increased change- and fault-proneness; and (4) refactor mutated classes that contain LAs to prevent the introduction of more faults in their system.

1.5 Roadmap

The remainder of this dissertation provides the following content:

Chapter 3 reviews related work on quality, linguistic anti-patterns, design patterns, and design anti-patterns quality analysis.

Chapter 2 provides the definition of linguistic anti-patterns, design patterns, and design anti-patterns, as well as the detection techniques that we used for our studies.

Chapter 4 reports our first study on the impact of linguistic anti-patterns on the quality and specifically on code understandability. We present and discuss the results of the empirical study.

Chapter 5 presents our second study on exploring the impact of design anti-patterns and linguistic anti-patterns on systems change- and fault-proneness, and provides quantitative evidence of the negative impact of design anti-patterns on classes change- and fault-proneness.

Chapter 6 investigates the mutation between design patterns and design anti-patterns in systems, and analyzes the effects of their mutation in classes on change- and fault-proneness. We also study the impact of the occurrences of linguistic anti-patterns on change- and fault-proneness during the evolution of systems.

Chapter 7 presents the conclusions of this dissertation and outlines some directions of future research.

CHAPTER 2 BACKGROUND

In this section, we provide background information about the different types of linguistic anti-patterns, design anti-patterns, and design patterns including their definitions, and detection.

2.1 Linguistic Anti-patterns

2.1.1 Definition

Linguistic anti-patterns refer to bad naming practices of identifiers, comments, and implementation of code entity. Arnaoudova *et al*. (2016) organized LAs (a.k.a lexical smells) into two categories: those where methods are affected and those where attributes are affected.. Table 2.1 summarizes these LAs providing their names, category, and short description. A detailed description of LAs is available in (Arnaoudova *et al*., 2016).

Table 2.1 List of Linguistic anti-patterns by (Arnaoudova *et al*., 2016).

	Name	**Category**	**Description**
A1	"Get" more than an accessor	Method	A getter that performs actions other than returning the corresponding attribute without documenting it.
A2	"Is" returns more than a Boolean	Method	The name of a method is a predicate suggesting a true/false value in return. However, the return type is not Boolean but rather a more complex type allowing, thus a wider range of values without documenting them.
A3	"Set" method returns	Method	A set method having a return type different than void and not documenting the return type/values with an appropriate comment.
A4	Expecting but not getting a single instance	Method	The name of a method indicates that a single object is returned, but the return type is a collection.
B1	Not implemented condition	Method	The comments of a method suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.
B2	Validation method does not confirm	Method	A validation method (e.g., name starting with "validate," "check," "ensure") does not confirm the validation, i.e., the method neither provides a return value informing whether the validation was successful, nor documents how to proceed to understand.
B3	"Get" method does not return	Method	The name suggests that the method returns something (e.g., name starts with "get" or "return"), but the return type is void. The documentation should explain where the resulting data are stored and how to obtain it.
B4	Not answered question	Method	The name of a method is in the form of predicate, whereas the return type is not Boolean.
B5	Transform method does not return	Method	The name of a method suggests the transformation of an object, but there is no return value and it is not clear from the documentation where the result is stored.
B6	Expecting but not getting a collection	Method	The name of a method suggests that a collection should be returned, but a single object or nothing is returned.
C1	Method name and return type are opposite	Attribute	The intent of the method suggested by its name is in contradiction with what it returns.
C2	Method signature and comment are opposite	Attribute	The documentation of a method is in contradiction with its declaration.
D1	Says one but contains many	Attribute	The name of an attribute suggests a single instance, while its type suggests that the attribute stores a collection of objects.
D2	Name suggests Boolean, but type does not	Attribute	The name of an attribute suggests that its value is true or false, but its declaring type is not Boolean.
E1	Says many but contains one	Attribute	The name of an attribute suggests multiple instances, but its type suggests a single one. Documenting such inconsistencies avoids additional comprehension effort to understand the purpose of the attribute.
F1	Attribute name and type are opposite	Attribute	The name of an attribute is in contradiction with its type as they contain antonyms. The use of antonyms can induce wrong assumptions.
F2	Attribute signature and comment are opposite	Attribute	The declaration of an attribute is in contradiction with its documentation. Whether the pattern is included or excluded is, thus, unclear.

2.1.2 Detection

Arnaoudova *et al*. (2016) proposed a tool called LAPD (Lexical Anti-Patterns Detection) to detect occurrences of LAs in software systems. It relies on the Stanford natural language parser((Toutanova

et Manning, 2000)) to identify the Part-of-Speech of the terms constituting the identifiers and comments and to make relations between those terms. We used LAPD for our first contribution presented in Chapter 5, because it was the most recent approach that deals with large number of LAs; it has a catalog of 17 LAs. The rationale and specifications of these LAs are detailed in (Arnaoudova *et al*., 2013). LAPD is available on-line[1] as an extension of CheckStyle.

We also implemented an extension of PMD[2] used for our second and third contributions presented in Chapters 4, and 6, respectively. We reused the same algorithms as Arnaoudova *et al.* to make the detection process faster. This extension is also available on-line[3]. PMD is an open source static Java code analyzer. It is a plug-in for IDEs, like Eclipse or jEdit. It is used to find bad programming practices that can reduce performance. It also detects copied–pasted pieces of code and other poor practices. It uses an Abstract Syntax Tree to identify occurrences of poor practices. We added new rules and rule-sets related to each type of LA to detect linguistic anti-patterns. These rules stem from the definitions of the LAs by (Arnaoudova *et al*., 2016).

Our PMD-extension have been implemented to detect LAs in methods, variables and parameters, and documentation. We used the Stanford parser to extract the POS of words and WordNet to obtain the meaning of the words in the documentation, the methods signatures, and the relations between the words (like antonyms or synonyms).

Because it is essential to ensure that detected instances of LAs were true instances, three researchers from the PolyMORSE research group manually verified every code snippet involved in the experiments to confirm the presence (respectively absence) of LAs in the systems.

2.2 Design Anti-patterns

The concept of design anti-patterns in object-oriented systems was first introduced by (Webster, 1995) in his book. Three years later, (Brown *et al*., 1998) introduced 40 types of design anti-patterns in terms of lower-level code smells. In the following, we present some of them.

2.2.1 Definition

Design anti-patterns are poor design solutions to recurring design problems. In this dissertation, we focus on 13 design anti-patterns from Brown *et al*. (1998) and Soloway *et al*. (1983). The motivation behind our choice is that these design anti-patterns have been thoroughly described by Brown *et al*. (1998) and they have received significant attention from researchers, e.g., (Khomh

[1] http://www.veneraarnaoudova.com/linguistic-anti-pattern-detector-lapd/
[2] https://pmd.github.io/
[3] http://www.ptidej.net/downloads/replications/sqj19a/Tool/

et al., 2012), (Taba *et al*., 2013). We could detect several occurrences of these design anti-patterns across the studied releases, and they are representative of design and implementation problems related to object-oriented systems.

- AntiSingleton (AS): a class that provides mutable class variables, which could be used as global variables.
- Blob (BL) or God Class (GC): a class that is too large and not cohesive enough, which monopolizes most of the system processing, takes most of the decisions, and is associated to data classes.
- ClassDataShouldBePrivate (CS): a class that exposes its fields, thus violating the principle of encapsulation.
- ComplexClass (CC): a class that has (at least) one large method and complex method, in terms of cyclomatic complexity and line of codes LOCs.
- LargeClass (LC): a class that has (at least) one large method, in terms of LOCs.
- LazyClass (LZC): a class that has few fields and methods and doesn't do enough.
- LongMethod (LM): a class that has a method that is overly long, in terms of LOCs.
- LongParameterList (LP): a class that has (at least) one method with a long list of parameters with respect to the average numbers of parameters per methods.
- MessageChain (MCh): a class that uses a long chain of method invocations to realize one of its functionality.
- RefusedParentBequest (RP): a class that overrides methods using empty bodies.
- SpaghettiCode (SC): a class declaring long methods which do not have any parameters. The class uses the many gotos, exceptions, global variables, or any other unstructured constructs. These methods have too complex and complicated control structures. This class does not use polymorphism and–or inheritance.
- SpeculativeGenerality (SG): a class that is defined as abstract but that has very few children, making use of its methods.
- SwissArmyKnife (SA): a class whose methods can be divided into disjoint sets of many methods, thus providing many different, unrelated functionalities.

2.2.2 Detection

We use the Defect DEtection for CORection Approach (DECOR) proposed by (Moha *et al.*, 2010) to detect occurrences of design anti-patterns. DECOR offers a domain-specific language to automatically generate design defect detection algorithms.DECOR leverages the Patterns and Abstract-level Description Language meta-model (PADL) (Guéhéneuc et Antoniol, 2008) and the Primitive, Operators, and Metrics framework (POM) (Gueheneuc *et al.*, 2004) to detect design anti-patterns in object-oriented systems.

A domain-specific language is more flexible than ad hoc algorithms (Moha *et al.*, 2010) because, domain experts (software developers) can modify the detection rules manually using high-level abstractions, considering the context, environment, and characteristic of the analyzed systems. PADL (Guéhéneuc et Antoniol, 2008) is a meta-model to describe object-oriented systems at different abstraction levels while POM (Gueheneuc *et al.*, 2004) is a PADL-based framework that implements more than 60 metrics. The output of DECOR is a list of classes and their roles (if any) in occurrences of design anti-patterns.

Using this domain-specific language, DECOR proposes the descriptions of several design anti-patterns. It also provides algorithms and a framework, DeTeX, to convert design anti-pattern descriptions automatically into detection algorithms. DeTeX allows detecting occurrences of design anti-patterns in various object-oriented systems written in different programming languages, such as Java or C++. We used DECOR because it has been widely-acknowledged and used in past and recent research; it achieves 100% recall while having a 31% precision rate in the worst case; with an average precision greater than 60%.

2.3 Design Patterns

A design pattern is a general repeatable solution to a commonly occurring design problems. Design patterns are reported to improve software quality. We study eight design patterns from Gamma *et al.*'s book, i.e., (Gamma, 1995), in our study listed below. We selected theses specific patterns because of their popularity and because previous works also studied them, e.g., (Tsantalis *et al.*, 2006; Vlissides *et al.*, 1995).

2.3.1 Definition

The complete definition and specification of all design patterns are available in previous work (Vlissides *et al.*, 1995; Khomh *et al.*, 2009a). In the following, we define only the ones considered in this thesis.

- Builder (Bu): a pattern to separate the construction of a complex object from its representation.
- Command (Cm): a pattern to encapsulate a request as an object.
- Composite (Cp): a pattern that composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator (De): a pattern that attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.
- Factory Method (FM): a pattern that defines an API for object creation in which subclasses choose the class to instantiate.
- Observer (Ob): a pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Prototype (Pt): a pattern that specifies the kind of objects to create using a prototypical instance.
- Singleton (Si): a pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

2.3.2 Detection

We use the Design Motif Identification Multi-layered Approach (DeMIMA) by (Guéhéneuc et Antoniol, 2008) to detect occurrences of design patterns. DeMIMA checks the traceability between design motifs (the micro-architecture describing the solutions of the design patterns) and the source code. The tool discovers idioms relevant to binary class relationships and then provides an idiomatic model of the source code. The model helps to identify design motifs while making a design model for the system. DeMIMA can recover idioms related to both the relationships among classes and design motifs describing the organization of the classes. DeMIMA specifies use, association, aggregation, and composition relationships in the system.

DeMIMA is organized in three layers. The first two layers are used to recover the abstract model of the source code including the binary class relationships, while the third layer defines design motifs and their representations in the abstract model.

DeMIMA uses explanation-based constraint programming to identify occurrences of design motifs using the roles and, relationships describing the motifs and the PADL models of the systems. It

reports the micro-architectures that are occurrences of the motifs as well as approximations done during the identification process. The output of DeMIMA is a list of classes and their roles (if any) in the occurrences of design patterns.

Guéhéneuc and Antoniol (Guéhéneuc et Antoniol, 2008) report that DeMIMA achieves 100% recall and 34% precision when detecting design patterns.

CHAPTER 3 RELATED WORK

In this chapter, we present a survey of related work on smells, their evolution, and impact on change- and fault-proneness. We categorized these studies into linguistic anti-patterns, design anti-patterns, and design-pattern detection and their evolution and impact on change- and fault-proneness.

3.1 Definition and Detection of Linguistic Anti-patterns (LAs)

Multiple studies on source code identifiers, e.g., (Caprile et Tonella, 2000; Merlo *et al*., 2003; Caprile et Tonella, 1999; Anquetil et Lethbridge, 1998) have highlighted the necessity of choosing appropriate and meaningful identifiers during the implementation of a software system. In this section, we summarize previous work related to the definition of LAs, their detection, and their effect on software evolution.

Abebe *et al*. (2009) proposed the first definition of lexicon bad smells. These lexicon bad smells are characterized by the use of abbreviations, contractions, and–or strange grammatical structures. They report that such lexicon bad smells have a negative impact on concept location activities. Lexicon bad smells can lead to understandability issues during software maintenance, which can be solved through refactoring. Following this pioneer work, Arnaoudova *et al*. (2013) defined a new category of smells (i.e., LAs) related to poor practice on naming, documentation, and implementation of an entity. They focused on a higher level of details related to inconsistencies between method names, parameters, return types and comments, and also between attribute names, types, and comments. They categorized LAs into two groups based on methods and attributes.

There are different proposed approaches to detect lexicon bad smells and LAs in particular. Tan *et al*. (2007) proposed three useful approaches to detect incoherence between code and comments. The first called, @iComment, detects lock- and call- related inconsistencies (Tan *et al*., 2007). The second approach, @aComment, find synchronization incoherence relevant to interrupt context (Tan *et al*., 2011). Finally, @tComment, is an automatic approach which proposed to deduce the properties form Javadoc linked to null values and exceptions; it performs test case generation when there is an inconsistency between the obtained properties (Tan *et al*., 2012).

Besides, De Lucia *et al.* proposed COCONUT to verify consistency between the lexicon of high-level artifacts and of source code, based on the textual similarity between the two artifacts (De Lucia *et al*., 2011). The tool is very useful to improve the quality of identifiers and comments.

In 2013, Abebe et Tonella (2013) built an ontology to assist developers in the choice of identi-

fiers consistent with the concepts already used in the system. Finally, as a more recent work by (Arnaoudova *et al*., 2016) proposed an approach called LADP to identify inconsistencies among identifiers, source code, and comments. This technique handles generic naming and comments issues in object-oriented programs, specifically in the lexicon and comments of methods and attributes. The tool is very useful to detect LAs on c++ and Java source codes, but it is somehow slow for large systems. We used this tool for two of our studies.

3.2 Definition and Detection of Design Anti-patterns (DAPs)

In 1995, (Webster, 1995) wrote the first book on "anti-patterns" in object-oriented systems. The book describes design anti-pattern as a frequently used solution to a problem that provides ineffective effect.

One year later, Riel (1996) proposed 61 heuristics description of good object-oriented programming to assess a program quality manually and improve its design and implementation. These heuristics are similar to code smells.

Two years later, Brown *et al*. (1998) discussed about 40 types of design anti-patterns in terms of lower-level code smells which, are considered as the basis of all the approaches to detect design anti-patterns.

In this area, several approaches have been proposed to detect design anti-patterns. (Van Emden et Moonen, 2002) developed the JCosmo tool which is able to visualize the code layout and design anti-patterns locations. The tool uses primitives and rules to detect the presence of smells and design anti-patterns while parsing the source code into an abstract model (similar to the Famix meta-model). The goal of JCosmo is evaluating code quality and helping developers to do refactorings. The main strength of JCosmo is visualizing problems by picturing the design.

Marinescu *et al.* developed a set of detection strategies to identify design anti-patterns using metrics (Rapu *et al*., 2004). Then, they added information collected from the documentation of problematic structures to the previous proposed detection strategies to detect design anti-patterns.

(iPlasma) (Marinescu et Lanza, 2006) is a unique platform for Software Modeling and Analysis to detect design anti-patterns which calculates metrics from C++ or Java source code and applies rules. The rules combined the metrics to find code fragments.

Khomh *et al*. (2009b) proposed a new approach using Bayesian network, which improve design anti-pattern detection. Later on, Settas *et al*. (2012) built on this idea and proposed another approach that quantify the existence of a design anti-pattern based on probabilistic knowledge about them. This knowledge contains the relationships of design anti-patterns through their causes, symp-

toms and consequences.

In this thesis, we used the DECOR method for design anti-patterns detection proposed by (Moha *et al*., 2010). We selected thus tool because it has been used in past and recent research; it achieves 100% recall while having a 31% precision rate in the worst case; with an average precision greater than 60%. We explain the approach in more details in Chapter 2.2.2.

3.3 Definition and Detection of Design Patterns (DPs)

Gamma (1995) wrote the first book on "design patterns" in object-oriented development.

Besides, one of the first papers about design patterns detection is by (Krämer et Prechelt, 1996). They introduced an approach to detect design information directly from C++ header files and stored in a repository. The design patterns are represented as PROLOG rules which are used to query the repository. They try to detect five structural design patterns namely Adapter, Bridge, Composite, Decorator, and Proxy.

Vokáč (2004) proposed an approach based on similarity scoring to compute the similarity between the graph of a design pattern and the graph of a system to identify classes participating to a design pattern.

There is another study by (Iacob, 2011) that identified proven solutions for recurring design problems using design workshops and systems analysis. It means that, during a design workshop, a team of 3-5 designers design a system while considered design issues are collected. Moreover, a set of systems are analyzed in order to recognize how the design issues are considered in the implementation of existing solutions.

In this thesis, we applied DeMIMA by (Guéhéneuc et Antoniol, 2008) to detect the design patterns presented in Chapter 2.3.2.

3.4 Evolution and Impact of Linguistic Anti-patterns

Abebe *et al*. (2012) examined the benefits of using LAs information alongside structural metrics in fault prediction models. They measured the accuracy of prediction models in two scenarios: (1) only structural metrics and (2) structural metrics and LAs. Through a case study with three open-source systems (*i.e.,* ArgoUML, Rhino, and Eclipse), they reported a significant improvement in fault prediction capability, for models using LAs information.

Abebe *et al*. (2011) suggested an approach to reveal the extent to which LAs can hinder the execution of maintenance tasks. Using this approach, they conducted a study and reported that LAs negatively affect concept location when using IR-based techniques.

Fakhoury *et al*. (2018) investigated the impact of the quality of lexicons on developers' cognitive loads using a functional brain-imaging technique and an eye-tracker. They provided empirical evidences of the negative impact of poor source code lexicon on developers' cognitive loads. Cognitive load is related but not identical to understanding. It is a proxy measure for a subset of understandability, focusing on effort.

We agree with the above-mentioned works that design anti-patterns are indicators of poor code quality and that LAs can hinder program understanding and the execution of maintenance tasks, as well as decreasing the quality of programs. In our work, we explore and compare the impact of different types of LAs on the time and effort required to complete some understandability tasks, as well as the correctness of the tasks. We also empirically investigate the additional relationship that linguistic anti-patterns can have with change- and fault-proneness.

3.5 Evolution and Impact of Design Anti-patterns and Design Patterns

There are few studies that investigated both design anti-pattern and design-pattern evolution. Bieman *et al*. (2003) claimed that there is a relative stability in design pattern classes compared to other classes. They showed that large classes are the most change-prone while pattern-based classes are more change-prone than other regular classes.

Vokáč (2004) reported that different design patterns affect fault-proneness differently. This result was obtained by studying a large C++ industrial system.

In the same direction, Gatrell *et al*. (2009) demonstrated that pattern-based classes are more change-prone than non-pattern classes.

Olbrich *et al*. (2009) examined historical data of Lucene and Xerces over several years and found that Blob classes and classes subjected to Shotgun Surgery are more change-prone than other classes.

Khomh *et al*. (2012) investigated the impact of anti-patterns on the change- and fault-proneness on classes. They considered 13 design anti-patterns and analyzed 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino. They observed that classes participating in design anti-patterns are significantly more likely to be changed than other classes. This study also investigated two types of changes experienced by classes with design anti-patterns: structural and non-structural changes. Structural changes can change the class interface while non-structural changes concern only method bodies. They concluded that structural changes are more likely to occur in classes participating in design anti-patterns.

Yamashita et Moonen (2013) reported that developers cannot fully evaluate the overall maintain-

ability of a software system considering code smells definitions alone. Therefore, they recommend combining different analysis approaches to achieve more complete and accurate evaluations of the overall maintainability of a software system. In the same direction, (Taba *et al.*, 2013) claimed that design anti-patterns can tell developers about design choices which could be a good or bad one. They try to improve the accuracy of fault prediction models using various metrics based on design anti-patterns. To achieve this goal, they considered the history of design anti-patterns from their inception. The results suggest that files participating in design anti-patterns are more fault-prone than the other files. Besides, the proposed anti-pattern-based metrics can improve fault prediction models, helping developers to focus their testing activities on classes that are more likely to exhibit faults.

Jaafar *et al*. (2013) examined relationships between design anti-patterns and design patterns. Their study showed that some design anti-patterns are significantly more likely to have relationships with design patterns than others.

Jaafar *et al*. (2014) in a more recent work investigated the mutation of design anti-patterns and its impact. They concluded that design anti-patterns often mutate into another form of more complicated design anti-patterns. However, they found that mutated anti-pattern classes are significantly less fault-prone than non-mutated classes.

In our work, We investigate how design patterns and–or anti-patterns mutate over time by modeling the behavior of these mutations using Markov models. We investigate the impact of these mutations on fault-proneness. We also study the types of changes to the code that lead to design patterns and anti-patterns mutations. And finally, We study the most fault-prone transitions from design patterns and–or anti-patterns.

CHAPTER 4 LINGUISTIC ANTI-PATTERNS AND PROGRAM COMPREHENSION

Linguistic anti-patterns negatively impact the understandability.
Having knowledge about LAs, and being proficient in the
language has a positive impact on understandability.

In this chapter, we perform an empirical study to investigate the impacts of linguistic anti-patterns on code understandability.

4.1 Context

Design anti-patterns are "poor" practices in the design, documentation, or implementation of software artifacts. They are usually introduced by developers who are not familiar with the software system at hand and–or do not have enough knowledge and experience in solving some particular problems (Brown *et al*., 1998; Khomh *et al*., 2012). Previous works reported that anti-patterns have a negative impact on program comprehension (Abbes *et al*., 2011). Hofmeister *et al*. (2017) also showed that shorter identifier names impact program comprehension negatively.

4.1.1 Research Problem and Contribution

Linguistic anti-patterns (LAs) introduced by (Arnaoudova *et al*., 2016) are anti-patterns that relate to inconsistencies in naming, documentation, and implementation of code entity. Arnaoudova *et al*. (2016) examined the developers' perception of the quality of code snippets containing LAs and reported that developers consider LAs to be poor practices that should be refactored. Fakhoury *et al*. (2018) studied the effect of LAs and readability on developers' cognitive load using a minimally-invasive functional brain-imaging technique (functional Near Infrared Spectroscopy, fNIRS) and an eye-tracker. They reported that LAs have a negative impact on developers' cognitive load. Although these previous works established a relation between the occurrence of LAs and program comprehension, they did not investigate if different types of LAs affect comprehension differently. Moreover, they did not study the effect of prior knowledge of the LAs on developers' understanding of code containing LAs.

In this chapter, we perform an empirical study to investigate how the occurrences of LAs in the systems impact on code understandability as one of the main factors of software quality.

To achieve this goal, we report on two experiments *ExpBefore* and *ExpAfter* investigating whether seven different types of LAs affect code understandability equally. These two experiments allow us

to also investigate whether knowing the characteristics of LAs improves developers' understandability of code snippets containing LAs.

In the first experiment (*i.e., ExpBefore*), participants do not have any knowledge of the LAs. We ask the participants to perform program comprehension tasks on code containing LAs or the refactored code without LAs. After this experiment, we give participants a series of lectures about LAs. We follow the lectures by a test aimed at assessing their acquired knowledge of LAs. Two weeks after these lectures, we conduct a second experiment (*i.e., ExpAfter*) using different systems and different comprehension tasks but under the same conditions as *ExpBefore*. We measure the participants' performance in the two experiments using three metrics: (1) perceived effort (measured using the NASA Task Load Index, TLX), (2) time spent to find correct answers to the tasks (measured using a hidden automatic timer), and (3) correctness of their proposed solutions (measured by comparing the solutions to an oracle). A total of 230 participants from two Universities participated in our experiments. We retained and analyzed the responses of 142 participants who completed all the experiments and quiz, and who did not have any prior knowledge about LAs.

Our results show that the presence of one LA in a code snippet affects the participants' correctness, time, and effort, considerably. Specifically, the LAs *"Set method returns"*, *"Says one but contains many"*, and *"Not answered questions"* have the most negative impact on understandability while *"Says many but contains one"* and *"Attribute name and type are opposite"* are the most difficult to detect. It also reports that knowing about LAs yields considerable improvements in the participants' correctness and a modest improvement in their time and effort. Therefore, it may be beneficial to teach developers about LAs. Also, being fluent in the language in which comments and identifiers are written can mitigate the negative impact of LAs.

4.1.2 Research Questions

This chapter answers the following research questions:

- **RQ4.1:** ***Do LAs affect developers' understandability of the code?*** To answer this research question, we ask the participants to perform two comprehension tasks and six detection tasks, using different code snippets from the studied systems, that may or may not contain LAs. We examine whether the occurrence of LAs in the code affects the time spent by participants performing the tasks, their effort, and the correctness of their solutions.
- **RQ4.2:** ***Do different types of LAs affect unknowledgeable developers' understandability?*** We distinguish between different types of LAs. For each of the seven types of LA under study, we define one comprehension or detection task, and study whether different types of

LAs affect the participants' understanding differently when the participants do not know LAs (ExpBefore).

- **RQ4.3:** ***Do different types of LAs affect knowledgeable developers' understandability?*** We repeat the same experiment as in **RQ4.2** but after giving a series of lectures about LAs to the participants. Thus, we investigate whether knowing LAs changes the impact of LAs on the participants' correctness, effort, and time, in comparison with the results of **RQ4.2**.
- **RQ4.4:** ***Can knowledge about LAs mitigate the impact of LAs on understandability?*** We investigate if having knowledge about LAs helps participants in understanding code snippets when performing comprehension and detection tasks. We evaluate the participants' knowledge about LAs through a quiz. Then, we investigate potential correlations between their level of knowledge of LAs and the participants' understanding (measured in terms of time, effort, and correctness).
- **RQ4.5:** ***Can knowledge of the language in which comments and identifiers are written mitigate the effect of LAs on developers' understandability of the code?*** LAs are related to the meaning of identifiers and the comments written in the code. We hypothesise that participants fluent in the language in which the identifiers and comments are written (*e.g.,* English) should have less difficulty when faced by LAs in comparison to those who are not fluent in that language. We compare the performances of participants who are fluent in English with that of others who are not fluent.

4.2 Study Design

We now describe our studied LAs, experiment design, participants, analyzed systems, and independent and dependent variables.

4.2.1 Studied LAs

Arnaoudova *et al*. (2016) examined LAs from seven open source systems written in Java and C++, to understand how they are perceived by developers. They collected the opinion of both internal developers (who wrote the code) and external developers who had no prior knowledge of the studied systems. Table 4.1 summarises their findings. Building on these findings, we selected the following LAs for our study:

- A2: "Is returns more than a Boolean"
- A3: "Set method returns"

- B4: “Not answered question”
- F1: “Attributes name and type are opposite”
- F2: “Attributes signature and comments are opposite”.

We chose these LAs because they were deemed the least acceptable by the developers. In addition to these five LAs, we also selected the top-two most frequent LAs from our ten subject systems. The stacked chart in Figure 4.1 shows the percentages of each type of LA in the studied system. It indicates that D1–“Says one but contains many” and E1–“Says many but contains one” are the two most frequent LAs in all the studied systems. Figure 4.2 presents the most prevalent LAs in the studied systems. In the following, we present some examples of the seven selected LAs. More examples of LAs are available in (Arnaoudova *et al.*, 2013).

- D1: “Says one but contains many”.
- E1: “Says many but contains one”

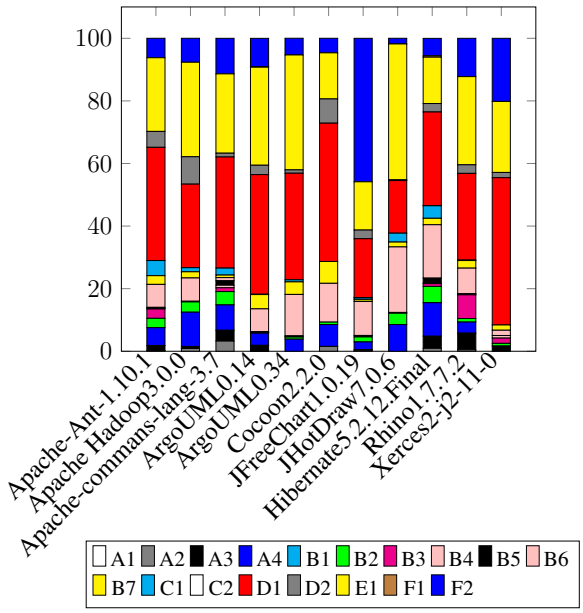

Figure 4.1 LAs occurrences percentages in the studied systems

Listing 4.1 Example of A2–“Is returns more than a Boolean” (Cocoon2.2.0)

```
public int isValid () {
    final long currentTime = System.currentTimeMillis();
    if ( currentTime <= this . expires ) {
```

Table 4.1 Detected occurrences of LAs studied in (Arnaoudova *et al.*, 2016) and developers' perceptions

	Name	External Developers' perception			Internal Developers' perception	Detected LAs	
		Very Poor	Poor	Total		Java Systems	Whole studied systems
F2	Attribute signature and comments are opposite	%30	%63	%93	It is a poor practice	%0.79	%2.16
B4	Not answered question	%34	%48	%82	It is a poor practice	%1.19	%0.45
F1	Attribute name and type are opposite	%23	%54	%77	It is a poor practice	%0.03	%5.37
A3	"Set" method returns	%43	%32	%75	It is a poor practice	%14.39	%6.36
A2	"Is" returns more than a Boolean	%13	%47	%60	It is a poor practice	%1.09	%2.44
A4	Expecting but not getting a single instance	%7	%30	%37	It is a poor practice	%5.58	%2.30
B3	Get method does not return	%36	%54	%90		%2.80	%0.87
C2	Method signature and comment are opposite	%41	%41	%82		%11.78	%8.51
B6	Expecting but not getting a collection	%14	%66	%80		%7.76	%3.37
D2	Name suggests Boolean but type are opposite	%20	%57	%77		%7.26	%8.30
E1	Says many but contains one	%14	%62	%76		%10.63	%17.65
B1	Not implemented condition	%44	%24	%68		%10.76	%3.37
B2	Validation method does not confirm	%14	%54	%68		%9.34	%6.16
C1	Method name and return type are opposite	%14	%54	%68		%0.03	%0.26
D1	Says one but contains many	%11	%29	%40		%10.73	%23.7
B5	Transform method does not return	%4	%54	%58		%4.95	%4.05
A1	"Get" more then an accessor	%4	%32	%36		%0.86	%0.64

```
    // The delay has not passed yet --
    // assuming source is valid .
        return SourceValidity.VALID;
    }
    // The delay has passed,
    // prepare for the next interval .
    this.expires = currentTime + this.delay;
    return this.delegate.isValid();
}
```

Listing 4.1 shows an example of "A2–"Is" returns more than a `Boolean`" LA, which belongs to the subcategory of "Methods". The method name `isValid` suggests that the method returns a `boolean` but it returns more information, which is counterintuitive. The LA occurs in class `DelayedValidity`. A proper documentation should provide explanation about the lack of returned value.

Listing 4.2 Example of A3–"Set method returns" (Apache Ant 1.10.1)

```
public Object setProperty(final String key, final String value) throws NullPointerException {
    final Object obj = super.setProperty(key, value);
    // the above call will have failed if key or value are null
    innerSetProperty(key, value);
    return obj;
```

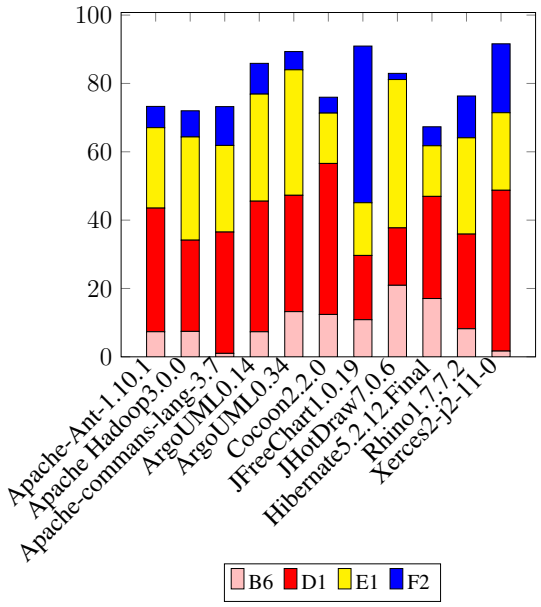

Figure 4.2 The most prevalent LAs in the studied systems

```
}
```

Listing 4.2 presents an example of LA of type “A3”, where the method name is `setProperty` but it returns an Object. Setters should not return anything.

Listing 4.3 Example of B4–“Not answered question” (ArgoUML v0.34)

```
protected void hasEditableBoundingBox (boolean value){
    bboxField.setEnabled(value);
    bboxLabel.setEnabled(value);
}
```

Listing 4.3 presents an example of the “B4–Not answered question" LA.
The method name `hasEditableBoundingBox` has the form of a predicate but its return type is not `Boolean`. The method `hasEditableBoundingBox` declared in class `StylePanelFig` has a name that suggests a `Boolean` value but it returns nothing (`void`).

Listing 4.4 Example of D1–“Say one but contains many” (Rhino1.7.7.2)

```
// constructor
public TableModelCritics () {}

// accessors
```

```
public void setTarget(vector critic){
  _target = critic;
  // fireTableStructureChanged();
}
```

Listing 4.4 shows an example of LA of type "D1". There is an attribute name `critic` that is singular but the type is `vector`.

Listing 4.5 Example of E1–"Says many but contains one" (Hibernate-release-5.2.12-final)

```
public class Cashing{
// NOTE: TruthValue for now because I need to look at how JPA's SharedCacheMode concept is
    handled
  private TruthValue requested = TruthValue.UNKNOWN;
  private String region;
  private AccessType accessType;
  private boolean cacheLazyProperties;

  public Cashing(TruthValue requested){
      this.requested = requested;
  }

}
```

Listing 4.5 contains a LA of type"E1", where the name of attribute is `cashLazyProperties` while the type is `Boolean`. The name of attribute suggests that it stores a collection but the type is Boolean.

Listing 4.6 F1–Example of "Attributes name and type are apposite" (ArgoUML 0.14)

```
        MAssociationEnd Start = null;
```

Listing 4.6 presents a LA of type "F1". The name of an attribute from class `ActionNavigability` uses the antonyms start and end, as a part of the type of the attribute.

Listing 4.7 F2–Example of "Attributes name and type are opposite" (ArgoUML 0.14)

```
public class SuffixLines
  extends BaseParamFilterReader
  implements ChainableReader{
```

```
    // Parameter name for the prefix.
    private static final String SUFFIX_KEY = ''suffix'';

    // The suffix to be used.
    private String suffix = null;

    //Data that must be read from, if not null.
    private String queuedData = null;

    // Constructor for ''dummy'' instances.
    public SuffixLines() {
        super();
    }
    ...
}
```

Listing 4.7 presents a LA of type "F2". An attribute name in class `SuffixLines` is suffix while the comment documenting it says `Parameter name for the prefix`.

4.2.2 Experiment Design

We perform two experiments to evaluate the impact of LAs on participants' understandability. We ask the participants to perform program comprehension tasks on code containing or not occurrences of LAs. All the experiments were conducted throughout a period of six weeks. The first experiment was performed by participants who did not have any knowledge about LAs (ExpBefore) during one week. Afterwards, these participants were educated about LAs. After this course, they took a quiz test about LAs within a period of one week. To investigate whether knowing LAs can improve understandability in terms of time, effort, and correctness, two weeks later, the same participants had one week to perform the second experiment (ExpAfter). For each step (experiments and quiz), we asked participants to complete all the tasks within the same day of their choice.

We evaluated the participants' knowledge about LAs through a quiz after a dedicated course to ensure they understood LAs well. If the quiz revealed that some participants did not understand the LAs well, then we would exclude them from our collected data. A total of 230 students attended our lectures on LAs and took the quiz test.

We divided the students to two groups, Group A and Group B, based on their group IDs.

In ExpBefore, we provided two similar on-line questionnaire forms for two groups. Group A are

participants with odd student IDs while Group B are those who have even student IDs. Each group had eight questions related to different types of LAs. ExpAfter has the same structure as ExpBefore, but different code snippets, and different questions. Having two groups of students help to have more questions and cover more types of LAs.

Each questionnaire contains four questions: Q1 and Q2 are general comprehension questions on excerpts from the studied systems; Q3 is a detection question in a Java class; Q4 includes five sub-questions Q(4.a to 4.e) to ask participants to find possible LAs in code snippets and to suggest the refactoring solutions. We give Q1 to Group A with one particular type of LA while Group B is given a refactored version of the same source code (i.e., with the LA removed). We do the contrary for Q2 in which Group B receives the LA and Group A has the refactored code. We want to investigate the impact of the occurrences of LAs on code understandability, where each of the two groups deal with the code that contain one type of LA, and the clean code on another question.

For ExpBefore, in the first question (Q1), we asked participants to find the method name that does a specific task to understand whether bad method naming can impact their understandability, and for the second question (Q2), we asked them a similar question, but now focusing on variables, to understand how bad attribute naming can affect their understandability. Since we focus on understandability, we asked participants in Q3, which element(s) were responsible for the confusion (in case they reported to have been confused by some identifiers and–or comments). Q(4.a to 4.e) were about finding bad coding practices in some code snippets and comments. We wanted to investigate the extent to which participants could identify LAs in a piece of code.

In ExpAfter, for both questions (Q1, and Q2), we asked comprehension questions on code containing “A3”, and “B4”, to understand the impact of these two LAs on code understandability? Q3, Q(4.a to 4.e) were similar to those asked in ExpBefore.

Figure 4.3 and Table 4.2 present the design of this study, in which “Correct” indicates the code snippets not containing any LA. The questionnaires for ExpBefore, Quiz, and ExpAfter are available on- line[1].

As it can be seen on Table 4.2, we did not have the same number of answers per LA in ExpBefore and ExpAfter. For example, Group A in ExpBefore who answered a question about “B4” had 40 participants, while in ExpAfter, Group B which dealt with the same LA (i.e., “B4”) had 50 participants. Because of these differences between the size of our corresponding groups (i.e., Group A in ExpBefore with 40 participants versus Group B in ExpAfter with 50 participants), we took average values for time, effort, and correct answers in each steps, to make the results comparable.

For each research question, we also removed outlier values, because we observed the same trend

[1] http://www.ptidej.net/downloads/replications/sqj19a/Questionnaire/

when considering or not these outliers. We assumed that the outliers are due to participants who did not consider the time, *i.e.,* they opened a question and took a break. Indeed, participants did not know that the time was recorded because the timer was hidden. Overall, each participant performed at least one task for each type of LA.

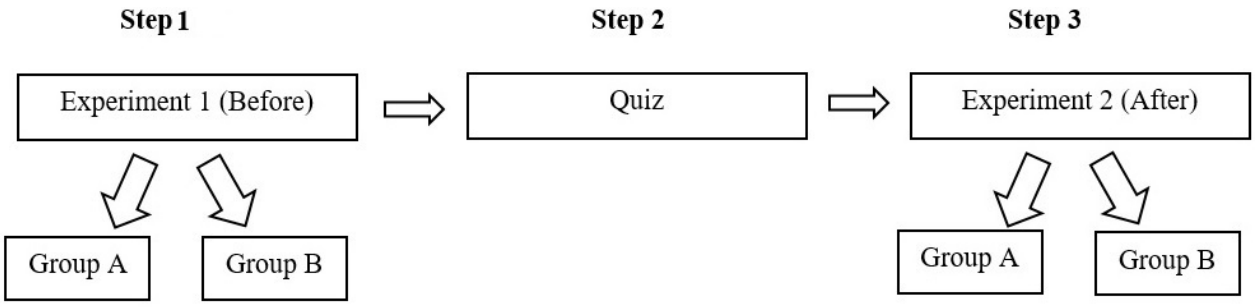

Figure 4.3 *Study Design Diagram*

Table 4.2 Study Design

Question	**ExpBefore**		**ExpAfter**	
	Group A	Group B	Group A	Group B
Q1	A2 (with)	A2 (without)	A3 (with)	A3 (without)
Q2	E1 (without)	E1 (with)	B4 (without)	B4 (with)
Q3	F1	F1	D1	D1
Q(4.a)	Correct	A3	E1	E1
Q(4.b)	B4	D1	F2	F2
Q(4.c)	Correct	Correct	F1	F1
Q(4.d)	F2	Correct	A2	A2
Q(4.e)	A3	F2	A3	A3

4.2.3 Participants

A total of 230 participants were involved in our two experiments. These participants were graduate students from Polytechnique Montreal and Concordia University. The participants from Polytechnique Montreal were students in the courses LOG8430 (Software Architecture and Advanced Design) and LOG8371 (Software Quality Engineering), while those from Concordia University were enrolled in the course SOEN6461 (Software Quality Engineering). All these participants reported that they have good programming skills. A majority of them had between one and three years of experience working in Industry prior to embarking in graduate studies. From our initial group of

230 participants, only 142 people had no prior knowledge of LAs and completed all our activities, i.e., the two experiments (ExpBefore and ExpAfter), the lectures, and the quiz test. Therefore, in the following, we report only the results of these 142 participants. The drop-out rate of 38% is due to the voluntary nature of the experiments (i.e., the students were not forced to participate in the experiments). Also, we excluded all the students who reported to have a good knowledge of LAs because we wanted to know if training developers about LAs could improve their ability to deal with code containing LAs (we needed participants with no prior knowledge of LAs in ExpBefore).

4.2.4 Studied Systems

Using convenience sampling as (Shull *et al.*, 2007), we chose ten systems written in Java from different domains, and different sizes for the experiments: Apache Ant, Apache Hadoop, Apache commons-lang, ArgoUML (two versions), Cocoon, JFreeChart, JHotDraw, Hibernate, Rhino, and Xerces. These systems are open source; allowing for an easy replication of our study, which have been studied several times in the previous studies. In definition, Apache ANT[2] is a system related to software build processes. Apache Hadoop[3] includes a collection of open-source software utilities using a network of many computers to solve data and computation problems. Apache commons-lang[4] is used to provide extra methods, which can not be produced by the standard Java libraries for manipulation of its core classes. ArgoUML[5] is an open-source UML modeling tool written in Java. Cocoon[6] is a web application framework based on the concepts of pipeline, separation of concerns and component-based web development. JFreeChart[7] is an open-source framework use to create a wide variety of both interactive and non-interactive charts. JHotDraw[8] is a two-dimensional graphics framework for structured drawing editors that is written in Java. Hibernate (ORM)[9] is an open-source Java persistence framework project. Rhino[10] is a commercial 3D computer graphics and computer-aided design application software, and Xerces[11] is a collection of software libraries for parsing, validating, serializing and manipulating XML.

First, we detect LAs in these systems to find the most frequent LAs. Second, we use different Java classes and code snippets from these systems as real examples of the occurrences of LAs. Table 4.3 describes the studied systems.

[2] http://ant.apache.org/
[3] https://hadoop.apache.org/
[4] https://commons.apache.org/proper/commons-lang/
[5] http://argouml.tigris.org/
[6] https://cocoon.apache.org/
[7] http://www.jfree.org/jfreechart/
[8] https://sourceforge.net/projects/jhotdraw/
[9] http://hibernate.org/
[10] https://www.rhino3d.com/
[11] https://xerces.apache.org/

We also chose these systems because previous studies analyzed some of them (i.e., “ArgoUML, Cocoon, JFreeChart, Xereces”) to study the impact of design anti-patterns on program comprehension (Abbes *et al.*, 2011) and that of linguistic anti-patterns on change- and fault-proneness (Guerrouj *et al.*, 2015). We also chose these systems because they contain different types of LAs and could be easier or more complex to understand. For ArgoUML, we considered two versions, because in ArgoUML0.14 there were a LA of type “F1”, but developers solved it during the software evolution. Other types of LAs were also removed during the evolution of ArgoUML0.14. Hence, we also study the last version of ArgoUML (ArgoUML0.34).

Table 4.3 Studied Systems

	Systems	Release date
System 1	ArgoUML0.34 ArgoUML0.14	2011-12-15 2003-12-05
System 2	Cocoon2.2.0	2013-03-14
System 3	JFreeChart1.0.19	2014-07-31
System 4	JHotDraw7.0.6	2011-09-06
System 5	Rhino1.7.7.2	2017-09-27
System 6	Xerces2-j2-11-0	2010-11-26
System 7	Apache Ant1.10.1	2017-02-06
System 8	Hibernate5.2.12.Final	2017-10-19
System 9	Apache commons-lang-3.7	2017-11-08
System 10	Apache Hadoop3.0.0	2017-12-13

4.2.5 Questions

We used comprehension and detection questions to collect data on the participants’ performances. Two researchers selected the Java classes and code snippets from the studied systems based on the studied LAs. Each Java class and code snippet contained only one type of LA. Refactored versions of these code snippets (containing no LA) were also produced. Similarly to (Abbes *et al.*, 2011), we considered questions in two of the four categories of questions regularly asked and answered by developers; see (Sillito *et al.*, 2008):

- Finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task;
- Focusing on a particular class believed to be related to some task and on directly related classes;

- Understanding a number of classes and their relations in some subsets of the source code;
- Understanding the relations between different subsets of the source code.

We only chose questions in the first two categories, because the last two categories concern questions that depend on different subsets of the source code and, in our experiments, LAs are located in a single subset of the code (i.e., they don't span multiple modules). For each group of participants, we defined the following two types of questions. The text in bold is a placeholder that we replaced by appropriate behaviors, concepts, elements, methods, and types depending on the systems on which the participants were performing their tasks.

- Category 1: Finding focus points:
  1. Which method is used to determine **this value**?
  2. How does this assignment affect **this variable**?

  For example, in the case of Apache Ant, we replace "this value" in Question 1, Category 1, by `restricted` and the question reads: "Which method is used to determine the value of the `restricted` attribute in the following link (Click on the link to show the code snippet)?"
- Category 2: Expanding focus points:
  1. Where is the bad coding practice (specifically LA) in **this method**?
  2. Where is the source of confusion in **this method**?

  In this category of questions (i.e., Category 2), we wanted to assess the participants' ability to recognize bad coding practices (i.e., LAs) in source code.

After designing the survey questionnaires, we conducted a pilot study with 5 students from our research group. This pilot study allowed us to refine the formulation of some questions. All the questions used in our experiments are available on-line at `http://www.ptidej.net/downloads/replications/sqj19a/Questionnaire/`.

4.2.6 Independent Variables

The independent variables of our experiments are variables capturing the presence/absence of the seven studied LAs, and a variable capturing whether participants acquired or not knowledge of LAs. These are variables that could affect the understandability of the participants.

4.2.7 Mitigating Factors

We have seven mitigating factors which could influence the participants' understanding in terms of time, effort, and correctness as follows:

- Participant's age
- Participant's gender
- Participant's degree
- Participant's knowledge level of programming
- Participant's knowledge level of Java
- Participant's working experience
- Participant's level of English

We collected these mitigating factors using a postmortem questionnaire (feedback). This feedback questionnaire was filled by participants at the end of ExpBefore. The questionnaire used a Likert scale for each of the mitigating factor mentioned above.

4.2.8 Dependent Variables

The dependent variables measure the participants' understandability in terms of effort spent doing the task, time spent to answer the questions, and percentage of correctness. We measure the participants' effort using the NASA Task Load Index (TLX) (Hart et Staveland, 1988). TLX is a multi-dimensional measure that evaluates the participants' subjective workload based on a weighted average of ratings on six sub-scales: mental demands, physical demands, temporal demands, own performance, effort, and frustration. We consider three sub-scales including "mental demands", "effort", and "frustration". NASA provides a computer program to collect weights of six sub-scales and ratings on these six sub-scales. We combine weights and ratings provided by the participants into an overall weighted workload index by multiplying ratings and weights; the sum of the weighted ratings divided by fifteen (sum of the weights) represents the effort (Hart et Staveland, 1988). An example of question asked to participants to capture "mental demand" is: "How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? We asked the participants to provide their responses on a scale from 1 (low) to 5 (high) for each question. We measured the time using a hidden timer in the questionnaires (ExpBefore and ExpAfter). The timer automatically started when the participant began performing

the comprehension task, and stopped when the participant finished the task, and submitted an answer. We received an automatic email whenever a participant completed a task. The email contains all their answers, the time they spent on the task, and their reported effort (from TLX). Finally, we computed the percentages of correctness for each question by dividing the number of correct answers by the total number of questions.

4.3 Study Results

In this section, we present the results obtained for each research question.

4.3.1 RQ4.1: Do LAs affect developers' understandability of the code?

To answer this question, we analyze the first two comprehension questions (Q1 and Q2) in the first experiment answers (ExpBefore). We have two groups of participants. Both groups deal with one code containing one LA and one code without any LAs. The LAs under study are: one LA from the methods category (A2 "Is returns more than a Boolean"), and one LA from the attributes category (E1 "Says many but contains one"). In the following, we analyze the participants' answers for each question in terms of time, effort, and correctness.

Correct Answers : For Q1, participants in the Group A were given code that contains one A2, while participants in Group B, answered questions on the refactored code (i.e., without any LA). Based on the collected data from Group A, and B, we can see that 39% of participants in Group B found the right answer, while in Group A, this percentage is 31%.

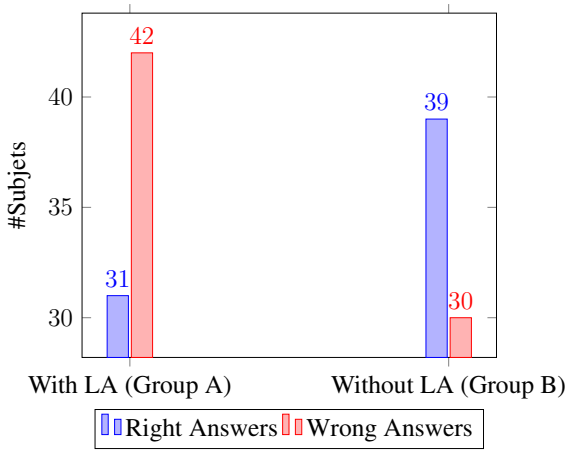

Figure 4.4 Impact of occurrence of LA on "Correctness":(Q1:A2)

For Q2, participants who were in the Group A were given the refactored code (i.e., without LA), while those in Group B answered the comprehension question on code containing an LA of type

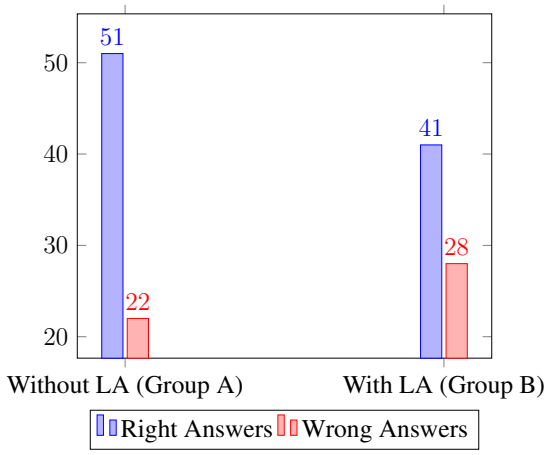

Figure 4.5 Impact of occurrence of LA on "Correctness":(Q2:E1)

E1. Based on the collected data, we observe that 51% of participants in Group A found the right answer, while in Group B, this percentage is only 41%. The sum of percentages of right and wrong answers in Figures 4.4 and 4.5 is not 100% because some participants did not complete the tasks. We did not want to consider abandons as wrong answers, since that would distort the results.

> Therefore, we can conclude that the occurrence of linguistic anti-pattern have negatively affected the understandability to a certain extent. Besides, different types of linguistic anti-pattern can have different effects on understandability. Based on the results, "A2" can have more negative impact on program comprehension in comparison with "E1".

Time: To answer this question, we just consider the time for the participants **who answered the question correctly**. Based on Figure 4.7, for Q1, it took 4.17 minutes on average for the participants in Group A who answered the question correctly, and Group B, answered in 2.06 minutes on average.

For Q2, it took 2.53 minutes on average for the participants in Group A to answer the question correctly, while for Group B, participants took 2.40 minutes on average; which is a bit less than for Group A.

For each research question, we removed outliers values, because we found the same trend when we consider outliers and when we removed them. We provide an example of both (with and without outliers) on Figures 4.6, 4.7, 4.8 and 4.9.

> Based on our observations, we can conclude that the occurrence of "A2" can reduce the speed of code understanding, but the occurrence of "E1" does not affect the time that much.

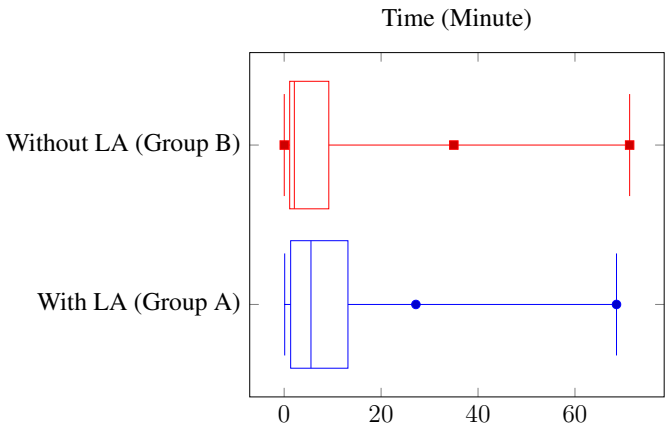

Figure 4.6 With Outliers

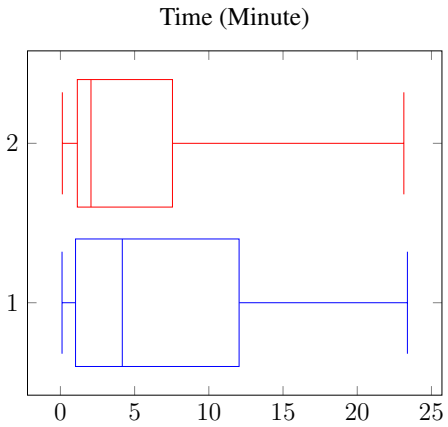

Figure 4.7 Impact of the occurrence of LA on "Time"(Q1): Without Outliers

Effort: Regarding participants self-reported efforts (captured using TLX), Figure 4.10 shows that for the first question, the participants in Group A who could answer the question correctly reported that finding the right answer was 60% difficult to some extent, while participants from Group B reported an effort to answer Q1 equal to 50%. For participants who answered incorrectly, the average effort reported was 46.6% in Group A, and 54.2% in Group B. Overall, participants who provided correct answers on systems containing the LA reported a higher effort than those who provided a correct answer on the system that didn't contain any LA. Regarding the effort reported by participants who failed to provide the correct answer, the effort is lower for those who worked on system containing the LA, which may be a sign that the inconsistencies of LAs may have mislead them in confidently picking a wrong answer.

For Q2, the participants in Group A who could answer the question correctly reported that finding the right answer was 53.3% difficult to some extend, while participants from Group B (who worked

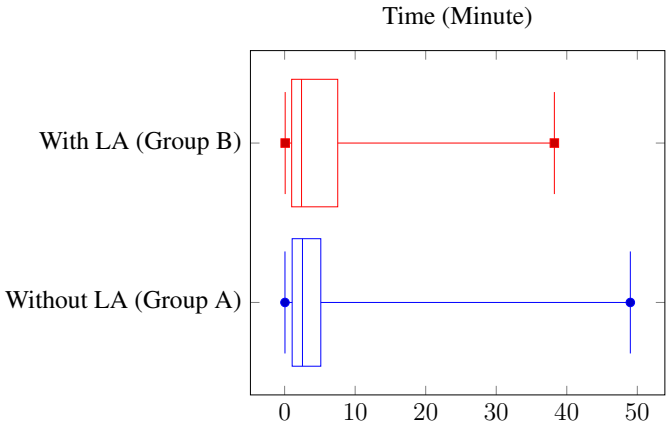

Figure 4.8 Impact of the occurrence of LA on “Time”(Q2): With Outliers

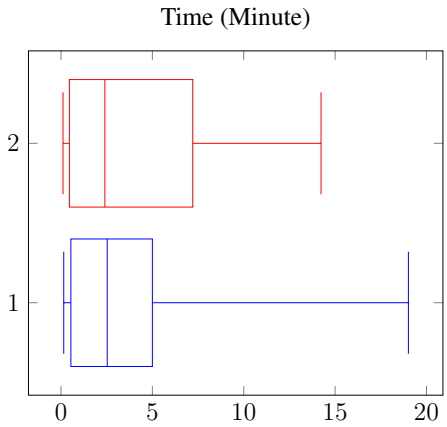

Figure 4.9 Impact of the occurrence of LA on “Time”(Q2): Without Outliers

on the system containing the LA) who provided the correct answer reported an average effort of 60%, as shown on Figure 4.11. This result is consistent with the result obtained for Q1. Participants from Group A who failed to provide a correct answer to Q2 reported an average effort of 48.1%, while those from Group B who failed to provide a correct answer reported an average effort of 61.6%. Overall, it seems that participants had a difficult time finding correct answers in the system containing the anti-pattern E1 “Says many but contains one”.

We performed the non-parametric Mann-Whitney test to compare the distributions of time, effort, and correct answers obtained from participants who answered the questions on code containing LAs versus those who worked on code without any LA. We did not find any statistically significant difference.

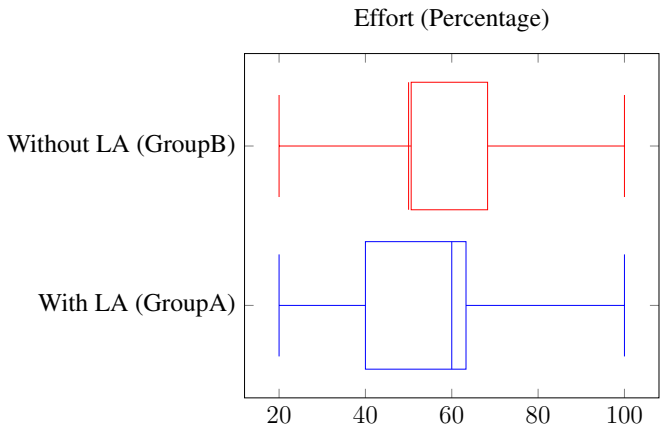

Figure 4.10 Impact of the occurrence of LA on “Effort”: (Q1)

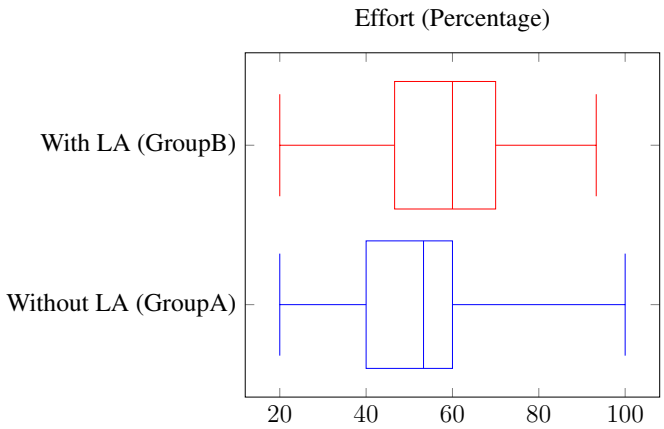

Figure 4.11 Impact of the occurrence of LA on “Effort”: (Q2)

Therefore, based on our observations, we can conclude that the occurrence of linguistic anti-pattern can increase the effort to understand the code to a certain extent. However, this increase is not statistically significant.

4.3.2 RQ4.2: Do different types of LAs affect unknowledgeable developers’ understandability?

In this research question, we analyze data obtained from ExpBefore. In ExpBefore, we investigated the impact of seven different types of LAs, using two metrics: correctness, and effort. We did not consider time when answering to this research question because, the level of complexity of the code snippets used in the different questions is not equivalent. Some LAs can appear in small pieces of code, while others can be present only in larger code snippets. Since it is expected that participants will take longer time inspecting/understanding a larger code snippet than a smaller code snippet. A

comparison of time would not have been relevant. Comparing the efforts and percentages of correct answers for different types of LAs allows us to identify LAs that have a more pronounced impact on understandability, as well as LAs that are difficult to identify in the code.

A2- "Is returns more than a Boolean". Participants from Group A in ExpBefore, answered the first question (Q1) including this type of LA. The results show that 31 participants out of 73, could answer the question correctly. Overall, they reported an average effort of 57.8%.

> A2- "Is returns more than a Boolean" is one of the easiest LAs to detect with an average effort of 57.8%.

A3- "Set method returns". We have one code snippet for this type in ExpBefore. Participants from both groups answered a question about a code snippet containing this type of LA. Group A have question (4.e), and Group B have question (4.a). The results show that only 6 participants from group A, and only one participant from Group B could detect A3 correctly. Besides, the effort is the highest value compared to other types. In general, we can conclude that when participants did not have knowledge about A3, this type of LA was the hardest to detect.

> A3- "Set method returns" has the most negative impact on understandability. Because, participants expect only a value assignment inside such methods, however there is a return value. In addition, it is hard to detect this LA according to participants' report. Practitioners should consider refactoring this type of LA as soon as possible.

B4- "Not answered question". In ExpBefore, only Group A in question (4.b) had a code snippet containing B4. 14 participants out of 73 detected it correctly. They reported an average effort of 63.3%, which is high in comparison to the other types.

> B4- "Not answered question" is one of the most problematic type, however, it has a less negative impact compared to "A3" and "D1" on program comprehension.

D1- "Says one but contains many". Only participants in the Group B from ExpBefore answered questions on code snippets containing D1, in question (4.b). The results show that only four participants out of 69 participants could detect this LA correctly. These four participants reported an average effort (to find D1) of 85%.

> D1- "Says one but contains many" is the second hardest LAs to detect among the other six types based on the proportion of correct answers. However, participants report that D1 is the hardest LA to detect in terms of effort.

E1- "Says many but contains one". In ExpBefore, the second question for Group B contains E1. In total 41 out of 69 participants could answer the question correctly. They reported an average effort of 56.5%.

> E1- "Says many but contains one" has been answered correctly by participants more than the other types of LA.

F1- "Attribute name and type are opposite". Participants from ExpBefore answered a question on systems containing F1 (question 3). The results show that among all our final 142 participants, only 26% (37 participants) of them identified this type of LA as a bad coding practice.

> F1- "Attribute name and type are opposite" is hard to detect according to the proportion of correct answers of participants.

F2- "Attribute signature and comments are opposite". In ExpBefore, participants answered questions on code containing instances of LA of type F2. Question (4.d) for Group A, and question (4.e) for Group B. In total, 25 participants from Group A, and 17 participants from Group B could detect F2 correctly among all the participants. They reported an average effort of 62.3%.

> F2- "Attribute signature and comments are opposite" is not very easy to find and detect when participants do not have knowledge about it. In addition, it is hard to detect this LA according to participants' reported effort.

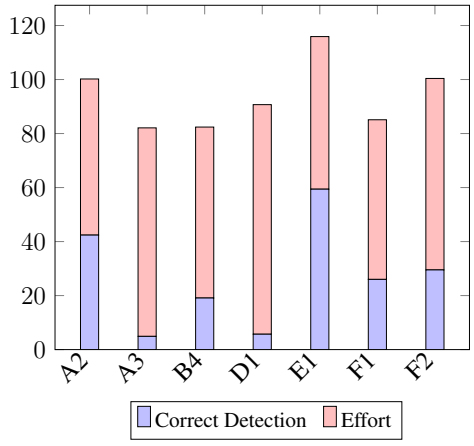

Figure 4.12 The most prevalent / least prevalent LAs in the studied systems (Before)

We Provide Table 4.4 to present all the obtained discussed results and make it easy to compare the impact of different LAs on participants' understandability before and after having knowledge about LAs.

4.3.3 RQ4.3: Do different types of LAs affect knowledgeable developers' understandability?

To answer this question, we do the same steps as in RQ4.2 but focusing on the data obtained from ExpAfter. Similar to **RQ4.2**, we only consider correctness and effort, to evaluate the participants' understandability. We want to know whether having knowledge about LAs can improve the participants' understandability? In the following, we discuss each type of LA in more details, following the same format as **RQ4.2**.

A2- "Is returns more than a Boolean". In ExpAfter, both groups of participants answered a question on code containing one A2 (Q4.d). The results show that 130 out of 142 participants could detect this LA. Participants reported an average effort of 61.4%.

> A2- "Is returns more than a Boolean" is one of the easiest LAs to detect when participants have knowledge of it. Participants reported an average effort of 61.2%.

A3- "Set method returns". In ExpAfter, both groups of participants had a question on code containing this LA (Q4.e). We obtained 134 correct answers out of 142 submitted responses. The mean reported effort is 62.2%. Group A also had the first question (Q1) on code containing this LA. We found that 72 participants answered correctly out of 73 submitted answers, with the average of 56.6%. Therefore, when combining the results of the two questions, we obtain that 95.8% of participants could do the task correctly, with an average effort of 59.4%.

> A3- "Set method returns" had the most negative impact on the understandability of developers who were not knowledgeable about it (see **RQ4.2**). The current results show that when participants have knowledge about this type of LA, they can detect it more easily. A prior knowledge of this LA also improved the correctness of their answers. However, the reported effort (i.e., 59.4%) suggests that despite the knowledge of the LA, developers still struggled a bit when performing their task on the code containing the LA. Understanding code containing inconsistencies in method returns (i.e., A3) can be hard. Practitioners should consider refactoring this type of LA as soon as possible.

B4- "Not answered question. In ExpAfter, only participants in Group B answered to question 2 which involved code containing B4. Almost all the participants were able to answer to the question correctly (67 out of 69), and the reported average effort is 62.8%.

In **RQ4.2**, only 14 participants out of 73 (19%) could identify B4- "Not answered question" correctly. This percentage increased to 97% after the participants were taught about the LA. Having a good knowledge of B4, seems to improve the understandability of code containing this LA.

D1- "Says one but contains many". All participants answered the question 3 containing D1 in ExpAfter. For this question 3, we obtained 68 correct answers out of 142 responses. Therefore, only 47.8% of participants provided the correct answers, even though they were taught about this LA prior to the experiment. The average reported effort is 66%.

D1- "Says one but contains many" is the hardest LA to detect among the seven studied LAs (based on the proportion of correct answers). The reported effort is also the highest (i.e., 66%). In **RQ4.2**, only 4 out of 69 participants successfully detected this LA. Although the proportion of correct answers improved after the participants were taught about the LAs, the effort required to deal with code containing the LA remained high.

E1- "Says many but contains one". In ExpAfter, all participants from both groups answered the question (4.a) which involved code containing an instance of E1. Among all the 142 participants, only 75 provided correct answers. They reported an average effort of 62.7%.

E1- "Says many but contains one" instances were not identified easily by the participants, even after learning about LAs. E1 is refer to the name of attributes based on their types. It seems that people usually do not consider such naming and they are not able to detect this type of LA easily, since it does not affect the code understandability.

F1- "Attribute name and type are opposite". In ExpAfter, participants answered question (4.c) which involved code containing F1. We obtained 69 correct answers out of 142 provided answers. The reported average effort is 60.6%.

Similar to E1, the participants had a hard time identifying instances of F1- "Attribute name and type are opposite", even after learning about the LAs.

F2- "Attribute signature and comments are opposite". In ExpAfter, participants answered to question (4.b) which involved a code snippet containing a LA of type F2. 132 out of 142 participants could find the correct answer. They reported an average effort of 62.6%.

F2- "Attribute signature and comments are opposite" is easy to find and detect when participants know about it.

Table 4.4 The impact of different LAs on the correctness, and effort (ExpBefore and ExpAfter)

Type	**Before**		**After**	
	Correctness(%)	Effort(mean)	Correctness(%)	Effort(mean)
A2	%42.4	%57.8	%91.5	%61.4
A3	%4.9	%77.2	%95.8	%59.4
B4	%19.1	%63.3	%97.1	%62.8
D1	%5.7	%85	%47.8	%66
E1	%59.4	%56.5	%52.8	%62.7
F1	%26	%59.1	%48.5	%60.6
F2	%29.5	%70.9	%92.9	%62.6

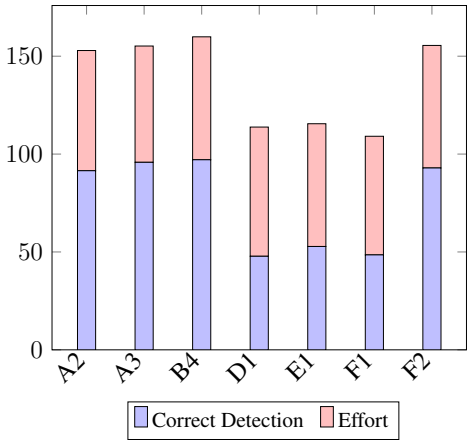

Figure 4.13 The most prevalent / least prevalent LAs in the studied systems(After)

As seen on Figure 4.12 and Table 4.4, the linguistic anti-patterns A3, D1, B4, and F1 have the most negative impacts on understandability. When the participants are educated about LAs, the impact is slightly mitigated (see Figure 4.13). Moreover, Figure 4.13 and Table 4.4 show that D1, E1, and F1 were still hard to detect by the participants after they learned about the LAs.

4.3.4 RQ4.4: Can knowledge about LAs mitigate the impact of LAs on understandability?

From **RQ4.3**, we observed that knowledge about LAs can mitigate their negative impact on understandability. In this research question, we want to know if the observed benefits in terms of understandability are proportional to the level of knowledge (of LAs) of the participants. To achieve this goal, we conducted a test quiz to capture the level of knowledge of the participants after the first experiment (i.e., ExpBefore) and the lectures about LAs. First we have to make sure that participants are knowledgeable about LAs. Therefore, after ExpBefore, we taught participants what

are the LAs, their consequences, and possible refactoring solutions. Then, we define some new questions as a quiz to evaluate the participants' knowledge. The quiz contain three first questions about the definitions of LAs and their impact on the quality. Then Q4 includes four code snippets that participate in one type of LA, and the participants need (1) to detect the LA, (2) explain the consequences of having this LA, and (3) provide solutions to refactor the problem.

Figure 4.15 presents the number of correct answers for each question of the quiz for Concordia University participants. In the quiz, questions 1, 2 and 3 are general definition questions and more than 93% of participants were able to answer these questions correctly. After these first 3 questions, participants were asked to detect LAs in four different code snippets containing respectively A2, A3, D1, and F1. As can be seen on Figure 4.15, more than 96% of participants were able to detect A2, A3, and F1 correctly. The participants were less successful in detecting D1, but still, almost 85% of participants answered its corresponding question correctly. At Polytechnique Montreal, more than 89% of participants were able to detect A2, A3, and F1 successfully (see Figure4.14). In general, all participants displayed a good knowledge of LAs after the lectures; the average percentage of correct answers is equal to 91%, which is considerable; showing that our teaching was efficient.

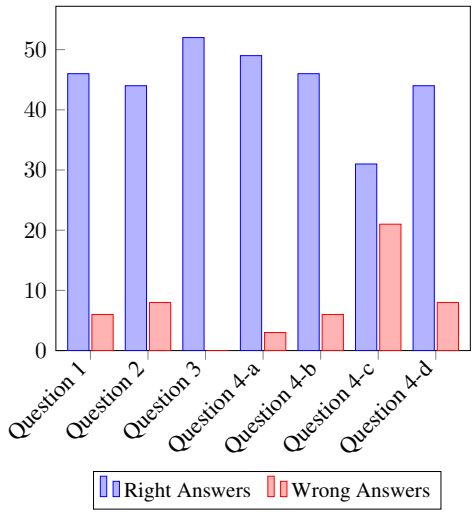

Figure 4.14 Right Answers (number): Polytechnique Montreal

Figure 4.16 presents quiz test results for participants of Polytechnique Montreal and Concordia University separately. Almost all the participants could answer more than 85% of the questions correctly.

Figure 4.17, 4.18, 4.19 presents the impact of having knowledge of LAs on the percentage of "correct answers", "Time", and "Effort". As one can see, the "Percentage of correct answers" is increased after learning the LAs; which is not surprising. In the case of "Time" and "Effort",

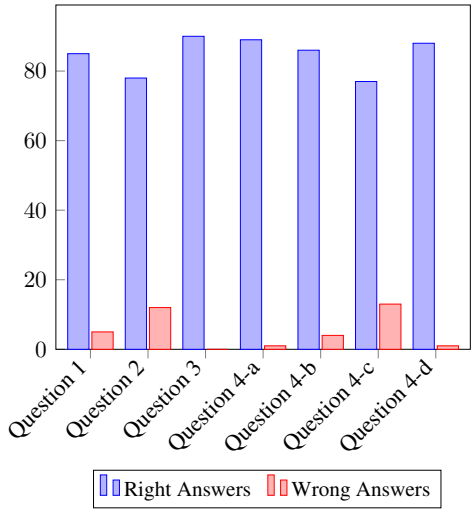

Figure 4.15 Right Answers (number): Concordia University

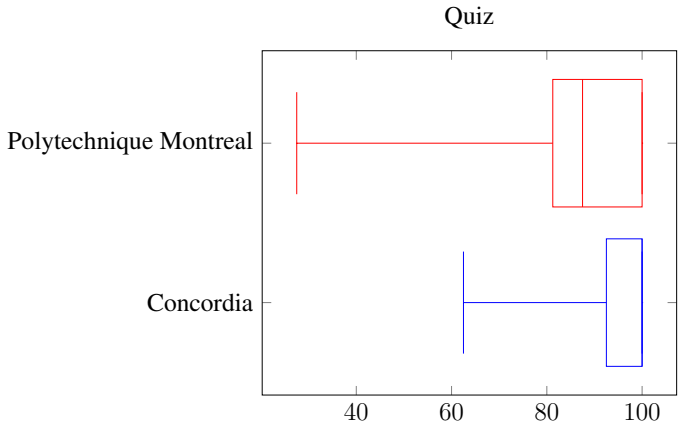

Figure 4.16 Evaluating the knowledge of LAs

Figures 4.17, 4.18, 4.19 show that there is no significant difference in terms of "Effort", while there is a slight improvement in term of "time" in average. Table 4.4 summarises experiments results obtained before and after training the participants. Overall, having knowledge of LAs positively affects the speed (and to some extent the effort) to find the right answers. We computed Spearman correlations between the score obtained by participants on the quiz test (capturing their level of knowledge of LAs) and respectively, the "Time" that they spent on the comprehension tasks, their percentage of "correct answers", and their reported "Effort".

We obtained a **weak** correlation of 0.24 between the quiz test scores and "Time"; a **very weak** correlation of 0.11 between the quiz test scores and the proportion of "correct answers"; and a **negligible** correlation of 0.07 between the quiz test scores and "Effort". We explain these results

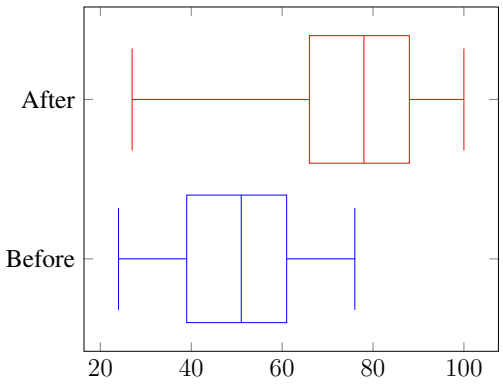

Figure 4.17 Impact of LA knowledge on Correct answers

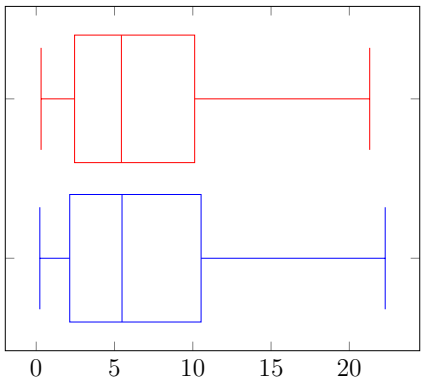

Figure 4.18 Impact of LA knowledge on Time(minutes)

by the fact that overall, all participants to experiment ExpAfter had acquired a very good knowledge of LAs, evident by their average quiz test score of 91%. The knowledge difference (i.e., about LAs) between the participants was small, hence the weak correlation with their performance.

We conclude that having knowledge about linguistic anti-pattern can mitigate the negative impact of LAs, increasing the speed at which developers comprehend the code containing these LAs.

4.3.5 RQ4.5: Can knowledge of the language in which comments and identifiers are written mitigate the effect of LAs on developers' understandability of the code?

LAs are bad coding practices related to the meaning of identifiers and the comments written in the code. We hypothesize that developers who are fluent in the language in which the identifiers and comments are written (e.g., English) should have less difficulty dealing with inconsistencies in identifiers and comments, in comparison to those who are not fluent in that language. During our experiments, we used a postmortem questionnaire to collect participant's age, gender, degree,

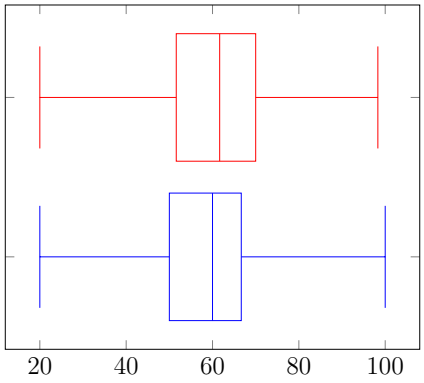

Figure 4.19 Impact of LA knowledge on Effort(Percentage)

knowledge level of programming, knowledge level of Java, working experience, and level of English. We use this information to perform correlations analysis, to understand the impact of participants' proficiency in English on code understandability since the code identifiers are written in English.

We applied an ANOVA test to evaluate the impact of "English" proficiency on the three measured variables (time, effort, and $\%$ of the correct answers). We used ANOVA to compare means between two or more groups of predictor variables. It can control the overall Type I error rate (i.e., false positive findings). Table 4.5 presents the obtained results. Overall, we find no statistically significant impact of participants level of English on code understandability.

However, when we divide our participants into two categories based on their main spoken language (Concordia participants are mostly English speakers while Polytechnique participants are French speakers), we observe some differences as shown in Table 4.6. In general, Concordia university participants have more correct answers than participants from Polytechnique Montreal. However, there is not much difference between the average time and effort of these two groups of participants. However, some other factors could impact on the understandability as well. For these experiments, Concordia participants are master and Ph.D students while the Polytechnique participants include undergraduate students. Besides, the programming experience could affect having more correct answers.

Table 4.5 p-values of the impact of English on dependent variables

University	**Correctness**	**Time**	**Effort**
Concordia University	0.29	0.81	0.97
Polytechnique Montreal	0.55	0.49	0.50

Table 4.6 Difference between English and French speakers on understandability

Activity	University	Correctness (Percentage)	Time (minute)	Effort (Percentage)
ExpeBefore	Concordia	%48.94	4.27	%60
	Polytechnique	%48.96	4.12	%58.33
ExpAfter	Concordia	%82.81	4.34	%61.11
	Polytechnique	%73.78	5.22	%56.66
Quiz	Concordia	%95.33	-	-
	Polytechnique	%86.05	-	-

In general, we can conclude that proficiency in the language in which identifiers and comments are written can have a slightly positive impact on understandability. However, this impact is not statistically significant.

4.4 Discussion

We now discuss our results and experiments.

4.4.1 Linguistic Anti-patterns

We studied seven different types of LAs. During the detection and the writing of the questions, we faced three problems regarding:

D1 – "Says one but contains many" we found a large number of real examples in our studied systems where this type of LAs occurs but is not conceptually an LA. We observed that developers prefer to choose "simple" names, like "tmp" for stack arrays, "x" and "y" for arrays of dimensions, or "v" for a collection of vectors. **Therefore, we decided not to identify such names as linguistic anti-patterns. Such cases should be considered as exceptions for this LA.**

E1 – "Says many but contains one" E1 is very similar to D1. When the type of an attribute is "int", our tool expected to have a singular name because "int" is a single type. Thus, it detected any plural names of type int as an occurrence of the E1 anti-pattern. However, we found that the names of these attributes could be plural because they are numerical variables that hold numbers of things, for example numbers of rows.

Here we provide one real examples:

Listing 4.8 Example of an exception of "E1" (JFreeChart 1.0.19)

```
// ** The maximum number of lines for category labels. */
```

```
private int maximumCategoryLabelLines;
```

Therefore, for numerical variables, plural names with singular type are meaningful and such naming should be considered as exceptions for this LA.

"Opposite meaning" we found a new type of LA, for example:

```
System.err.println("Computation successful");.
```

The statement uses the `err` attribute to print some error messages, yet the message is positive. **We propose to create a new LA that could be described as "the method name and its parameters are contradictory".**

4.4.2 Mitigating Factors

We also investigated to what extend other factors, like programming experience, gender, age, and education, could explain the participants' performance. We established a database containing information about these mitigating factors. We aggregated this data in sets for age (18-25, 26-35, and 36-45), gender (male, female), degrees (Bachelors or less, Masters, and PhDs or above), and working experience (less than 3 years, 3 to 5 years, 5 to 10 years, and more than 10 years).

We built a linear regression model to compute the impact of these factors on understandability, in terms of time, effort, and correctness. We applied an ANOVA test to evaluate the impact of these factors on time, effort, and correctness. Results show that participants' programming knowledge and education has a statistically significant impact on effort (given an α value of 0.05). Participants' education also has an statistically significant impact on the correctness of their answers, as shown in Table 4.7.

4.5 Threats to Validity

Construct Validity threats concern the accuracy of observations compare to the theory. We considered time, effort and the correctness of their answers to evaluate the participants' understanding. These measures are objective, but other factors, like fatigue, could change their values. We relied on the NASA TLX to measure the effort, which is inherently subjective because it is self-reporting. We accept the threat of self-reporting, future work should consider other measures of the effort, possibly through eye-tracking or other more invasive means.

Table 4.7 p-values of the impact of mitigating variables on dependent variables

Mitigating Variables	**Correctness**	**Time**	**Effort**
Age	0.13	0.08	0.52
Gender	0.21	0.69	0.92
Degree	**0.03**	0.41	**0.01**
Programming Knowledge	0.06	0.36	$<$ **0.01**
Working Experience	0.71	0.72	0.62
English	0.52	0.24	0.83
Knowledge	0.30	0.80	0.75

We used a quiz to evaluate the participants' knowledge about LAs after learning about these concepts. We wrote the quiz questions to measure the participants' knowledge but these questions could also have missed their objective. However, the results of the ExpAfter partially confirmed that participants acquired a good knowledge through our LA course and, thus, we accept this threat.

The impact of participants' experience as well as continuing education between the ExpBefore and ExpAfter could have had an impact on participants' understanding. Participants were full-time students taking several courses during their semesters. The two experiments took place during a whole semester. Therefore, there is the risk of participants gaining experience, which could affect their performance and, consequently, the results of this study. However, we claim that the impact, if any, should be low given the short time period between the two experiments, most likely not enough to bring participants from one level of experience to the next according to Dreyfus' model; (Dreyfus et Dreyfus, 1980).

We used the PMD-plugin to detect LAs by defining new rules related to each LA in Java systems. We accepted that the precision of this tool may not be 100%. Some false positive classes may have passed our manual validation if they "looked like" a LA. Conversely, our tool may miss some true positive classes. Moreover, in addition to the impact of LAs, other characteristics in the code could have influenced participants' understanding. We accept these threats because we built the code snippets manually and verified them and shared[12] them with the community for inspection and reuse.

Internal Validity threats concern confounding factors that might influence the results. We consider that four threats could impact the internal validity of our study.

-**Teaching** threats affect our second experiment, *i.e.,* ExpAfter, because we used data collected during this experiment (which occurred after a lecture) to evaluate the impact of participants' knowl-

[12] `http://www.ptidej.net/downloads/replications/sqj19a/`

edge on code understandability. We created two different groups of participants in each experiment to avoid bias (*e.g.,* gender bias). Each participant performed comprehension tasks on two different systems while the tasks for each system were different. Thus, we consider that the teaching effect is minimal because ExpAfter was performed purposefully after teaching participants about LAs.

-**Selection** threats concern the natural differences among the participants' abilities, which could affect our results. We examined the possible impact of these confounding factors using the metrics presented in Table 4.7, which shows that these factors did not significantly affect our dependent variables (*i.e.,* correctness, time, and effort), which thus mitigates this threat.

We agreed that there is no control in our experiments. because of the setting of a classroom, it is very hard to control the experiments. However, somebody could consider the course about LAs as an intervention and use Randomized Control Trial (RCT) as a type of scientific (often medical) experiment that aims to reduce certain sources of bias when testing the effectiveness of new treatments like learning process in this study. From the study presented in this chapter, we cannot claim causation between the occurrence of LAs in a system and code understandability. Although we found evidences of correlations, more analysis and experiments are required to strengthen our findings.

-**Instrumentation** threats were minimal considering that we used objective measures, such as time and correctness. Although there is some subjectivity in measuring participants' efforts using the NASA TLX. We believe the collected measurements reflect the actual feeling' of the participants regarding their effort and thus is an adequate proxy for their effort actual or perceived.

-**Diffusion** threats pertain to participants sharing tasks, questions, and answers. Although it is possible that a few participants exchanged some information, we defined two sets of questions to prevent too easy diffusion.

In addition, internal validity concerns our selection of the analyzed systems and analysis method. We defined some keywords for each question and made a query based on these keywords to compute the number of correct answers, time, and effort metrics for participants who answered the questions correctly. The accuracy of our defined keywords affects our results because the number of correct answers and consequently time and effort computed with these queries are used to analyze our data. To mitigate this threat, five researchers were involved in the identification of correct answers for each question.

Conclusion Validity threats concern the correctness of the results. We asked participants to answer detection and comprehension questions about LAs in code snippets instead of reading the whole code and doing maintenance tasks. Participants could have answered the questions differently if they were asked to do the maintenance tasks and were given the whole code. Due to the nature of

experiments with participants, we accept these threats as did many previous works.

We studied seven different LAs in each experiment and asked participants to perform tasks requiring time and effort. Thus, we opted for a simple design, which we believe to be realistic. We designed our questions related to the tasks to be simple while covering the requirements of our research questions. Moreover, this design is not unique, some previous studies (Hofmeister *et al.*, 2017; Fakhoury *et al.*, 2018) used the same design and code snippets to simplify their experiments.

Reliability Validity is about the possibility of replicating this study. We studied ten open-source systems with 230 participants but do not claim that these results are representative of all systems or developers. We provide all the necessary data on-line[13] to help other researchers replicate our work.

External Validity is related to the generalisability of the results. We performed our study on ten different, real systems belonging to different domains and with different sizes as shown in Table 4.3. We wrote questions such that participants needed to focus on some code snippets from the whole systems. We cannot claim that our results can be generalized to other systems written in other programming languages and to other participants. It is desirable that future works extend this study considering other LAs, other participants, other questions, and other systems.

4.6 Summary

This chapter examined the impact of occurrences of LAs on code understandability. We performed two experiments (ExpBefore and ExpAfter) to collect quantitative evidences about the impact of LAs on participants understanding of code snippets before and after a series of lectures on LAs and in function of their fluency in the natural language in which identifiers and comments are written.

Results showed that LAs negatively affect understandability. The LAs “Set method returns” (A3), “Not answered question” (B4), and “Says one but contains many” (D1) have the most negative impact on understandability, while “Is returns more than a Boolean” (A2) is the one with the least impact. Moreover, “Says many but contains one” (E1) and “Attribute name and type are opposite” (F1) are the most difficult to detect. Results also showed that having knowledge about LAs help participants to find correct answers faster. Proficiency in the language in which identifiers and comments are written had a slightly positive impact on understandability.

Thus, because LAs have a negative impact on code understandability, development teams should consider (1) educating their team members about LAs, (2) removing LAs from their software systems as soon as possible, and (3) using a common, well-known language for their identifiers and comments (not necessarily English).

[13] `http://www.ptidej.net/downloads/replications/sqj19a/`

In the next chapter, we study the impact of LAs on code quality in terms of change- and fault-proneness when there are also DAPs in the code. we further investigate the relation between design anti-patterns and LAs and their impacts on change- and fault-proneness.

CHAPTER 5 LINGUISTIC ANTI-PATTERNS, DESIGN ANTI-PATTERNS AND THEIR IMPACT ON CHANGE-, AND FAULT-PRONENESS

Linguistic anti-patterns can increase change- and fault-proneness on classes contain design anti-patterns as well. Classes containing only design anti-patterns are more change- and fault-prone than classes with linguistic anti-patterns only.

In this chapter, we perform an empirical study to understand the relation between linguistic anti-patterns and design anti-patterns, and their impact on software quality in terms of Change- and Fault- Proneness.

5.1 Context

Past and recent studies have shown that design anti-patterns (DAPs) which are poor solutions to recurrent design problems make object-oriented systems difficult to maintain, and that they negatively impact the class change- and fault-proneness, see (Khomh *et al.*, 2009a), (Abbes *et al.*, 2011). More recently, linguistic anti-patterns (LAs) have been introduced to capture recurring poor practices in the naming, documentation, and choice of identifiers during the implementation of an entity, see (Arnaoudova *et al.*, 2016), and (Arnaoudova *et al.*, 2013).

We analyze to what extent classes containing LAs have higher (or lower) odds to change or to be subject to fault-fixing than other classes containing DAPs.

5.1.1 Research Problem and Contribution

Some of the previous works (Taba *et al.*, 2013) (Yamashita et Moonen, 2013) reported that existing design anti-patterns in the systems lead to reproduce changes and fault-proneness somehow.

We believed that LAs could also impact on software systems' change- and fault-proneness. Therefore considering the case of classes which have both LAs and DAPs at the same time, we investigated their effect on increasing change and fault-proneness in comparison with the cases only having design anti-patterns or LAs.

5.1.2 Research Questions

To achieve the goal of the study, we defined two research questions as follows:

- **RQ5.1: Are classes with a particular family of smells (anti-patterns, linguistic anti-patterns, or both anti-patterns, linguistic anti-patterns) more change-prone than others?** To answer this question, we defined a hypothesis that is the proportion of classes undergoing at least one change between two releases is not different between classes containing different families of smells.

- **RQ5.2: Are classes with a particular family of smells (anti-patterns, linguistic anti-patterns, or both anti-patterns, linguistic anti-patterns) more fault-prone than others?** We considered another hypothesis to achieve the answer and test if the proportion of classes undergoing at least one fault-fixing change between two releases does not differ between classes with different families of smells.

5.2 Study Design

5.2.1 Studied Linguistic Anti-patterns and Design Anti-patterns

In this section, we focus on 12 design anti-patterns from (Brown *et al*., 1998) and (Soloway *et al*., 1983) described in section 2.2.1. The motivation behind our choice is that these design anti-patterns have been thoroughly described by Brown *et al*. (1998) and they have received significant attention from researchers (Khomh *et al*., 2012) (Taba *et al*., 2013). We could detect several occurrences of these design anti-patterns across the studied releases, and they are representative of design and implementation problems related to object-oriented systems. We studied 17 types of LAs which presented in Table 2.1 in chapter 2 2.1.1.

5.2.2 Experiment Design

We study changes and faults in a class whether they are relevant to the class containing a specific family of smells (e.g., LAs and DAPs) regardless of the kinds of smells from each family (e.g., Blob or LazyClass design anti-patterns). More precisely, we test whether the proportions of classes exhibiting (or not) at least one change/fault significantly vary between classes with 1) DAPs, 2) LAs, or 3) both. To address RQ5.1, we compute the following:

1. **#APs**: number of classes of a project release for which there was at least one class change and at least one design anti-pattern among the 13 design anti-patterns detected.

2. **#LAs**: number of classes of a project release for which there was at least one class change and at least one LA among the 17 LAs detected.

3. **#DAPs-LAs**: number of classes of a project release for which there was at least one class change and at least a design anti-pattern and a LA (both) among the 29 DAPs and LAs detected.

4. **#No-DAPs**: number of classes of a project release for which there was no DAP, while there was at least one class change.

5. **#No-LAs**: number of classes of a project release for which there was no LA, while there was at least one class change.

6. **#No-DAPs-LAs**: number of classes of a project release for which there was no DAPs and LAs at the same time, while there was at least one class change.

Then, we use the Fisher exact test (Sheskin, 2003) to assess whether the proportion between different families of smells significantly differs in terms of changes, faults. Fisher's exact test is used to determine whether there is a significant difference between two proportions or to test association between two characteristics. Fisher's exact test is appropriate for small samples. Specifically, we test whether the proportions between

1. Anti-patterns and LAs, i.e. the proportions (DAPs, No-DAps) and (LAs, No-LAs),

2. Both DAPs and LAs and DAPs (only), i.e. the proportions (DAPs-LAs, No-DAPs-LAs) and (DAPs, No-DAPs), as well as

3. Both DAPs and LAs and LAs (only), i.e. the proportions (DAPs-LAs, No-DAPs-LAs) and (LAs, No-LAs), show significant differences in terms of changes/faults.

As for RQ5.2, we compute the same proportions above, for the different considered families of design smells, but for faults (instead of changes) and then we assess whether the differences between the computed proportions significantly differs in terms of faults. We also use the Odds Ratio (OR) (Sheskin, 2003) as an effect size measure. Odds ratio indicates the likelihood of an event (i.e. change or fault) to occur. The OR is defined as the ratio of the odds p of an event occurring in one sample, i.e. the set of classes with one family of smells or both, i.e. LA, DAP, or LAs and DAPs (experimental group), to the odds q of it occurring in the other sample, i.e. the set of classes containing another different family of smells from the three investigated families, i.e. LA, DAP, or LA and DAPs (control group): $OR = \frac{p/(1-p)}{q/(1-q)}$ The interpretation of odds ratio is as follow. An odds ratio of 1 indicates that the event (i.e. change or fault) is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (experimental group) while an OR $<$

1 shows the opposite (control group). Since we perform several tests on the same data, we adjust p-values using the Bonferroni correction procedure (Sheskin, 2003). This procedure works as follow: it divides the critical p-value ($alpha$) by the number of comparisons, n, being made: $alpha/n$. In this study, we perform three pair of tests (e.g. design anti-patterns vs. LAs) when analyzing change- and fault-proneness, the null hypothesis is, therefore, rejected only if the p-value is less than 0.016 (0.05/3). We use Bonferroni because it is a simple procedure (Sheskin, 2003).

5.2.3 Studied Systems

We studied 30 releases of three randomly-chosen projects includes 7 releases of ANT[1] , 12 releases of ArgoUML[2] , and 11 releases of Hibernate[3]. In definition, ArgoUML is an open-source UML modeling tool, Hibernate (ORM) is an open-source Java persistence framework project while ANT is a system related to software build processes. We chose these systems because they belong to different domains and have different sizes5.1.

Table 5.1 Characteristics of the Analyzed Projects.

Projects	#Rel.	#Dev.	#Size (LOCs)	#All Classes	#Changes	#Classes Changed	#Faulty Changes
ANT	7	51	1,660,256	14,067	15,353	64,167	587
ArgoUML	13	25	644,829	27,822	5,300	23,153	201
Hibernate	10	89	7,239,075	21,876	9,075	89,658	179

The data collection process to storing the mentioned releases is the first step. Then, using version control systems; Git[4] for ANT and Hibernate, and SVN for ArgoUML; we able to mine the source code change history repository using SQL queries to identify source code change history, the information of the number of changes, classes that underwent changes, summary of the changes, change logs, etc and fault-fixes. The Git/SVN repository of each system was downloaded using appropriate perl scripts and the data was then stored in a PostgreSQL database. In the last step, we mined bug repositories corresponding to each system with the purpose of identifying changes that were fixing faults. For ArgoUML, issues dealing with fixing faults are marked as "DEFECT" in the issue tracking system[5]. For ANT, we mined BugZilla[6] while JIRA[7] was mined to determine fault-fixing issues for Hibernate. Finally, we use statistical tests to analyze the collected data and address our research questions.

[1] http://ant.apache.org/
[2] http://argouml.tigris.org/
[3] http://hibernate.org/
[4] http://git-scm.com/
[5] http://argouml.tigris.org/issues
[6] https://www.bugzilla.org/
[7] https://www.atlassian.com/software/jira

5.2.4 Identifying Post-Release Defects

In order to determine whether a change fixes a fault, we search, using regular expressions, in change logs from the system version controls; Git/SVN; for co-occurrences of fault identifiers with keywords like "fixed issue #ID", "bug ID", "fix", "defect", or "patch". We define post-release faults as in the study by Kamei *et al.* (Kamei *et al*., 2013) as those with fixes recorded in the six-month period after the release date. Once this step is performed, we identify, for each bug ID, the corresponding bug report from the corresponding issue tracking system, i.e., Bugzilla or Jira and extract relevant information from each report including:

- Issue ID.
- Issue type, i.e., fault, enhancement, feature, patch, feature request, etc.
- Issue status, i.e., new, closed, reopened, resolved, fixed, verified, or not.
- Issue resolution, e.g., fixed, invalid, duplicate, etc.
- Priority.
- Product name.
- Issue opening or closing dates.
- Issue summary. We do not use such info but we keep it in our database for possible further investigations.
- Data and name of issue reporter.

5.3 Study Results

In this section, we provide answers to our research questions.

5.3.1 RQ5.1: Are classes with a particular family of smells (anti-patterns, linguistic anti-patterns, or both anti-patterns, linguistic anti-patterns) more change-prone than others?

1. Classes containing both design anti-patterns(DAPs) and linguistic anti-patterns(LAs) vs. classes with design anti-patterns(DAPs)

For ANT, in all analyzed releases, Fisher's exact test indicates a significant difference in the proportion of changed classes between the group of classes containing in both DAPs and LAs and

those having DAPs only. Odds ratios vary across systems and, within each system, across releases. For ANT, we found an OR greater than 1 in all releases. The OR ranges from 1.98 (ANT 170) to 9.51 (ANT 180). This finding means, that for ANT, classes with both DAPs and LAs are more change-prone than classes containing DAPs only.
For ArgoUML, in 6 releases (out of a total of 13), Fisher's exact test indicates a significant difference in the proportion of changed classes between the group of classes with both DAPs and LAS and those containing DAPs only. Odds ratios vary across systems and, within each system, across releases. We find an OR greater than 1 for the release 0.18; this indicates that classes with both DAPs and LAS are more change-prone than classes with DAPs only. For the rest of releases, the OR is less than 1 and in few cases close to 1, i.e., the odd of experiencing a change is the same for classes with both DAPs and LAs and classes with DAPs only.
For Hibernate, we did not find any significant differences. Overall, we could not find that classes having both DAPs and LAs are more change-prone than classes containing DAPs only across all systems and releases.5.2

We therefore conclude that LAs do not increase the odds of a class to experience a change, i.e., they do not make classes with design anti-patterns more change-prone:

This finding brings empirical evidence on the fact that linguistic anti-patterns do not contribute to the change-proneness of design anti-patterns when both occur in classes of object-oriented systems.

2. Classes having design anti-patterns and linguistic anti-patterns vs. classes containing linguistic anti-patterns

Table 5.3 shows the difference in proportions between the change-proneness of classes with both DAPs and LAs and classes with LAs only. As it can be noticed, results vary depending on the system. However, in several cases, Fisher's exact test show significant differences with OR greater than 1. For ANT, the OR ranges between 2.33 (ANT 192) and 5.66 (ANT 15 MAIN) while for Hibernate the OR is higher, it is between 13.24 (Hibernate 3.6.1) and 112.42 (Hibernate 3.6.2) which means that the difference, in terms of changes, is really high. For ArgoUML, the OR is between 1.95 (ArgoUML 0.30.1) and 3.76 (ArgoUML 0.26.2); these results suggest that, in most cases, the odd of experiencing a change is higher for classes with both DAPs and LAs than it is for classes with LAs only.

We therefore conclude that classes with both DAPs and LAs are changed in greater proportion than classes having only in LAs. When comparing the Odd ratios of (design anti-patterns and linguistic anti-patterns vs. design anti-patterns) and (design anti-patterns and linguistic anti-patterns vs.

Table 5.2 Change-Proneness Results: Design Anti-patterns and Linguistic Anti-patterns vs. Design Anti-patterns (only).

Release	#DAP,LA	#DAP	#No-Smell	#No-DAP	Adj. p-value	OR
ANT 151	27	266	0	119	<**0.0001**	-
ANT 152	29	269	0	119	<**0.0001**	-
ANT 154	26	244	0	57	**0.012**	-
ANT 170	42	146	48	331	**0.0047**	1.98
ANT 180	93	357	5	183	<**0.0001**	9.51
ANT 192	83	292	14	198	<**0.0001**	4.01
ANT 15 (MAIN)	23	162	4	220	<**0.0001**	7.77
Hibernate 3.6.1	100	736	2	22	1	1.49
Hibernate 3.6.2	77	589	17	149	0.68	1.14
Hibernate 3.6.3	0	538	0	181	1	0
Hibernate 3.6.4	0	452	0	274	1	0
Hibernate 3.6.7	0	304	0	420	1	0
Hibernate 3.6.8	0	315	0	455	1	0
Hibernate 4.2.5	0	512	0	504	1	0
Hibernate 4.2.7	0	492	0	486	1	0
Hibernate 4.3.0	0	469	0	639	1	0
ArgoUML 0.14	24	365	36	471	0.68	0.86
ArgoUML 0.16	26	397	44	437	0.10	0.65
ArgoUML 0.18	41	514	44	1077	**0.003**	1.95
ArgoUML 0.18.1	43	576	30	201	**0.0083**	0.50
ArgoUML 0.20	41	459	46	364	0.14	0.70
ArgoUML 0.22	45	653	75	285	<**0.0001**	0.26
ArgoUML 0.24	48	496	74	483	0.02	0.63
ArgoUML 0.26	42	435	66	525	0.22	0.76
ArgoUML 0.26.2	50	606	42	328	0.052	0.64
ArgoUML 0.28	96	374	64	591	0.232	0.79
ArgoUML 0.28.1	38	540	81	418	<**0.0001**	0.36
ArgoUML 0.30	241	370	965	595	<**0.0015**	0.50
ArgoUML 0.30.1	231	520	88	445	<**0.0001**	0.36

linguistic anti-patterns), we observe that:

> The occurrence of design anti-patterns in a class that experienced a linguistic anti-pattern seems to have a stronger relationship with change-proneness than the occurrence of linguistic anti-pattern in a class that experienced a design anti-pattern.

3. Classes containing design anti-patterns vs. classes with linguistic anti-patterns Table 5.4 reports on the proportion of changed classes in the groups of classes experiencing DAPs only and classes experiencing LAs only. As it can be noticed, in most of the system's releases, Fisher's exact test indicates that the difference between the change-proneness of classes with DAPs only vs. classes with LAs only is statistically significant.

Except for ANT, in which the OR is less than 1 indicating that the proportion of classes having DAPs that changed, is lower than the proportion of classes with LAs that changed, the OR is greater than 1 for all other releases of the analyzed systems. It ranges between 2.47 (Hibernate 3.4.6) and

Table 5.3 Change-Proneness Results: Design Anti-patterns and Linguistic Anti-patterns vs. Linguistic Anti-patterns (only).

Release	#DAP,LA	#LA	#No-Smell	#No-LA	Adj. p-value	OR
ANT 151	27	58	0	13	**0.01**	-
ANT 152	29	57	0	13	**0.0093**	-
ANT 154	26	51	0	2	1	-
ANT 170	42	59	48	110	0.08	1.62
ANT 180	93	157	5	17	0.24	2.00
ANT 192	83	129	14	51	**0.011**	2.33
ANT 15(MAIN)	23	38	4	38	**0.0013**	5.66
Hibernate 3.6.1	100	157	2	354	<**0.0001**	112.42
Hibernate 3.6.2	77	131	17	385	<**0.0001**	13.24
Hibernate 3.6.3	0	209	0	309	1	0
Hibernate 3.6.4	0	208	0	312	1	0
Hibernate 3.6.7	0	63	0	461	1	0
Hibernate 3.6.8	0	60	0	468	1	0
Hibernate 4.2.5	0	29	0	1274	1	0
Hibernate 4.2.7	0	24	0	628	1	0
Hibernate 4.3.0	0	59	0	660	1	0
ArgoUML 0.14	24	26	36	58	0.28	1.48
ArgoUML 0.16	26	30	44	59	0.73	1.16
ArgoUML 0.18	41	50	44	84	0.12	1.56
ArgoUML 0.18.1	43	53	30	91	**0.0023**	2.45
ArgoUML 0.20	41	43	46	104	**0.0073**	2.14
ArgoUML 0.22	45	53	75	95	0.79	1.075
ArgoUML 0.24	48	62	74	131	0.22	1.36
ArgoUML 0.26	42	54	66	138	0.07	1.52
ArgoUML 0.26.2	50	69	42	219	<**0.0001**	3.76
ArgoUML 0.28	32	44	64	244	**0.00031**	2.76
ArgoUML 0.28.1	38	53	81	228	**0.0059**	2.01
ArgoUML 0.30	30	41	96	241	0.033	1.83
ArgoUML 0.30.1	38	51	88	231	**0.0091**	1.95

75.16 (Hibernate 3.6.1) for Hibernate and between 1.78 (ArgoUML 0.16) and 7.20 (ArgoUML 0.30) for ArgoUML.

This findings is very likely due to the fact that DAPs are, in general, well known and established than LAs, i.e. they change more often in comparison with LAs as developers know they need to change them. We therefore conclude that:

> Design anti-patterns contribute more to the change-proneness of linguistic anti-pattern classes than linguistic anti-patterns do to the change-proneness of design anti-pattern classes.

Overall, we reject the first hypothesis since, in most of the analyzed systems, there is a significant difference between the proportion of classes undergoing at least one change between two releases, for classes belonging to different families of smells.

Table 5.4 Change-Proneness Results: Design Anti-patterns vs. Linguistic Anti-patterns.

Release	#DAP	#LA	#No-DAP	#No-LA	Adj. p-val	OR
ANT 151	266	58	119	13	0.0328	0.50
ANT 152	269	57	119	13	0.044	0.51
ANT 154	244	51	57	2	**0.0044**	0.16
ANT 170	146	59	331	110	0.33	0.72
ANT 180	357	157	183	17	<**0.0001**	0.21
ANT 192	292	129	198	51	**0.0039**	0.58
ANT 15(MAIN)	162	38	220	38	0.25	0.73
Hibernate 3.6.1	736	157	22	354	<**0.0001**	75.16
Hibernate 3.6.2	589	131	149	385	<**0.0001**	11.58
Hibernate 3.6.3	538	209	181	309	<**0.0001**	4.38
Hibernate 3.6.4	452	208	274	312	<**0.0001**	2.47
Hibernate 3.6.7	304	63	420	461	<**0.0001**	5.28
Hibernate 3.6.8	315	60	455	468	<**0.0001**	5.39
Hibernate 4.2.5	512	29	504	1274	<**0.0001**	44.55
Hibernate 4.2.7	492	24	486	628	<**0.0001**	26.44
Hibernate 4.3.0	469	59	639	660	<**0.0001**	8.20
ArgoUML 0.14	365	26	471	58	0.027	1.72
ArgoUML 0.16	397	30	437	59	**0.0137**	1.78
ArgoUML 0.18	514	50	1077	84	0.25	0.80
ArgoUML 0.181	576	53	201	91	<**0.0001**	4.91
ArgoUML 0.20	459	43	364	104	<**0.0001**	3.04
ArgoUML 0.22	653	53	285	95	<**0.0001**	4.10
ArgoUML 0.24	496	62	483	131	<**0.0001**	2.16
ArgoUML 0.26	435	54	525	138	<**0.0001**	2.11
ArgoUML 0.262	606	69	328	219	<**0.0001**	5.85
ArgoUML 0.28	374	44	591	244	<**0.0001**	3.50
ArgoUML 0.281	540	53	418	228	<**0.0001**	5.54
ArgoUML 0.30	370	41	595	241	<**0.0001**	7.20
ArgoUML 0.30.1	520	51	445	231	<**0.0001**	5.28

5.3.2 RQ5.2: Are classes with a particular family of smells (anti-patterns, linguistic anti-patterns, or both anti-patterns, linguistic anti-patterns) more fault-prone than others?

1. Classes containing both design anti-patterns and linguistic anti-patterns vs. Classes with design anti-patterns

Table 5.5 summarises Fisher's exact test results and ORs. The differences in the proportions of classes that undergo fault-fixing changes is mostly significant for ANT with an OR greater than 1 varying from 2.18 to 2.76; indicating that in ANT, the fault-proneness of classes with both DAPs and LAs is higher than the fault-proneness of classes with DAPs only.

For the remaining systems, there is no statistically significant difference between the proportions of classes that underwent fault-fixing changes among the groups of classes with both design anti-patterns and linguistic anti-patterns and classes with design anti-patterns, suggesting that in general:

Table 5.5 Fault-Proneness Results: Design Anti-patterns and Linguistic Anti-patterns vs. Design Anti-patterns (only).

Release	#DAP, LA	#DAP	#No-Smell	#No-DAP	Adj. p-value	OR
ANT 151	17	183	10	202	0.16	1.87
ANT 152	16	158	13	230	0.17	1.78
ANT 154	18	154	8	147	0.10	2.14
ANT 170	55	191	35	286	**0.00029**	2.34
ANT 180	50	174	48	366	**0.00051**	2.18
ANT 192	57	189	40	301	**0.00029**	2.86
ANT 15(MAIN)	20	194	7	188	0.02	2.76
Hibernate 3.6.1	6	28	96	730	0.278	1.62
Hibernate 3.6.2	6	29	88	709	0.271	1.66
Hibernate 3.6.3	0	28	0	691	1	0
Hibernate 3.6.4	0	33	0	693	1	0
Hibernate 3.6.7	0	33	0	698	1	0
Hibernate 3.6.8	0	32	0	692	1	0
Hibernate 4.2.5	0	32	0	738	1	0
Hibernate 4.2.7	0	43	0	973	1	0
Hibernate 4.3.0	0	41	0	937	1	0
ArgoUML 0.14	6	79	53	573	0.83	0.82
ArgoUML 0.16	5	100	55	736	0.53	0.66
ArgoUML 0.18	7	110	63	724	0.57	0.73
ArgoUML 0.18.1	13	141	72	1450	0.053	1.85
ArgoUML 0.20	12	143	60	634	0.87	0.88
ArgoUML 0.22	15	163	63	660	1	0.96
ArgoUML 0.24	12	165	93	773	0.13	0.60
ArgoUML 0.26	16	188	72	791	0.88	0.93
ArgoUML 0.26.2	17	190	61	770	0.65	1.12
ArgoUML 0.28	14	194	78	740	0.22	0.68
ArgoUML 0.28.1	13	188	78	777	0.26	0.68
ArgoUML 0.30	14	188	105	770	0.045	0.54
ArgoUML 0.30.1	15	178	111	817	0.10	0.62

Linguistic anti-patterns do not make classes with design anti-patterns more fault-prone (than they already are).

2. Classes having design anti-patterns and Linguistic anti-patterns vs. Classes containing Linguistic anti-patterns

Table 5.6 shows significant differences for some releases of the three systems. For three releases of ANT, OR values are greater than 1 ranging between 1.96 (ANT 170) and 3.67 (ANT 15 MAIN). For two releases of Hibernate the ORs are almost equal to 5. These findings suggest that, for the mentioned releases, the odd of experiencing a fault-fixing change is higher for classes with both DAPs and LAs than for classes with LAs only. We therefore conclude that, in some cases:

The occurrence of design anti-pattern in a class that experienced a linguistic anti-patterns have a strong relationship with fault-proneness than, the occurrence of linguistic anti-patterns in a class that experienced a design anti-pattern.

Table 5.6 Fault-Proneness Results: Design Anti-patterns and Linguistic Anti-patterns vs. Linguistic Anti-patterns (only).

Release	#DAP, LA	#DAP	#No-Smell	#No-DAP	Adj. p-value	OR	
ANT 151	17	29	10	42	0.06	2.43	
ANT 152	16	26	13	44	0.12	2.06	
ANT 154	18	27	8	26	0.15	2.14	
ANT 170	55	75	35	94	**0.01**	1.96	
ANT 180	50	70	48	104	0.09	1.54	
ANT 192	57	77	40	103	**0.01**	1.90	
ANT 15(MAIN)	20	33	7	43	**0.007**	3.67	
Hibernate 3.6.1	6	7	96	504	**0.011**	4.48	
Hibernate 3.6.2	6	7	88	509	**0.007**	4.93	
Hibernate 3.6.3	0	4	0	514	1	0	
Hibernate 3.6.4	0	5	0	515	1	0	
Hibernate 3.6.7	0	5	0	519	1	0	
Hibernate 3.6.8	0	5	0	519	1	0	
Hibernate 4.2.5	0	5	0	523	1	0	
Hibernate 4.2.7	0	8	0	1295	1	0	
Hibernate 4.3.0	0	8	0	644	1	0	
ArgoUML 0.14	6	6	53	75	0.56	1.41	
ArgoUML 0.16	5	6	55	78	1	1.18	
ArgoUML 0.18	7	7	63	82	0.77	1.29	
ArgoUML 0.18.1	13	17	72	117	0.68	1.24	
ArgoUML 0.20	12	17	60	117	0.52	1.37	
ArgoUML 0.22	15	17	63	117	0.23	1.63	
ArgoUML 0.24	12	17	93	117	0.84	0.88	
ArgoUML 0.26	16	17	72	117	0.33	1.52	
ArgoUML 0.26.2	17	17	61	117	0.11	1.91	
ArgoUML 0.28	14	19	78	269	0.017	2.53	
ArgoUML 0.28.1	0.68	17	78	264	0.024	2.58	
ArgoUML 0.30	14	17	105	264	0.06	2.06	
ArgoUML 0.30.1	15	17	111	264	0.04	2.09	

This finding is likely due to the fact that fault-fixing changes related to classes with design anti-patterns cover (according to the fault-fixing change logs) a variety of types including implementation problems, features, API changes, bugs after modification, deployment, and Find Bugs reported problems while linguistic anti-patterns are mostly associated with formatting issues, identifier naming, data types, enumeration types, spelling, check style, etc. This justifies why anti-patterns boost faults rates that much.

3. Classes containing design anti-patterns vs. Classes with linguistic anti-patterns

Table 5.7 reports on the proportion of classes that underwent fault-fixing changes in the groups of classes experiencing DAPs only and classes experiencing LAs only. Except for ANT and ArgoUML 0.16 and 0.18, Fisher's exact test show significant differences with an OR greater than 1 in all studied cases. For Hibernate, the OR ranges between 2.75 and 8.75 while it varies between 1.78 and 7.20 for ArgoUML.

This result suggests that:

> The occurrence of design anti-patterns in a class has a strong relationship with the class's fault-proneness than the occurrence of linguistic anti-patterns.

We reject the second hypothesis since in most cases there is a significant difference between the proportion of classes undergoing at least one fault-fixing change between two releases, for classes belonging to different families of smells.

Table 5.7 Fault-Proneness Results: Design Anti-patterns vs. Linguistic Anti-patterns.

Release	#DAP	#LA	#No-DAP	#No-LA	Adj. p-val	OR
ANT 151	183	29	202	42	0.36	1.31
ANT 152	158	26	230	44	0.59	1.16
ANT 154	154	27	147	26	1	1.08
ANT 170	191	75	286	94	0.36	0.83
ANT 180	174	70	366	104	0.05	0.70
ANT 192	189	77	301	103	0.32	2.84
ANT 15(MAIN)	194	33	188	43	0.25	1.34
Hibernate 3.6.1	28	7	730	504	**0.01**	2.75
Hibernate 3.6.2	29	7	709	509	**0.0089**	2.97
Hibernate 3.6.3	28	4	691	514	**0.00041**	5.20
Hibernate 3.6.4	33	5	693	515	**0.000169**	4.89
Hibernate 3.6.7	33	5	698	519	**0.00016**	4.90
Hibernate 3.6.8	32	5	692	519	**0.0002**	4.79
Hibernate 4.2.5	43	8	973	1295	<**0.0001**	7.14
Hibernate 4.2.7	41	8	937	644	**0.00052**	8.75
Hibernate 4.3.0	40	8	1068	711	**0.00084**	3.32
ArgoUML 0.14	365	26	471	58	0.027	1.72
ArgoUML 0.16	397	30	437	59	**0.0137**	1.78
ArgoUML 0.18	514	50	1077	84	0.25	0.80
ArgoUML 0.18.1	576	53	201	91	<**0.0001**	4.91
ArgoUML 0.20	459	43	364	104	<**0.0001**	3.04
ArgoUML 0.22	653	53	285	95	<**0.0001**	4.10
ArgoUML 0.24	496	62	483	131	<**0.0001**	2.16
ArgoUML 0.26	435	54	525	138	<**0.0001**	2.11
ArgoUML 0.26.2	606	69	328	219	<**0.0001**	5.85
ArgoUML 0.28	374	44	591	244	<**0.0001**	3.50
ArgoUML 0.28.1	540	53	418	228	<**0.0001**	5.54
ArgoUML 0.30	370	41	595	241	<**0.0001**	7.20
ArgoUML 0.30.1	520	51	445	231	<**0.0001**	5.28

5.4 Threats to Validity

Construct validity threats concern the relation between theory and observation. A main threat is related to the techniques used to detect DAPs and LAs. We applied DECOR (Moha *et al.*, 2010) for the identification of anti-patterns since it has been widely used in previous studies on

anti-patterns, while we applied LADP to detect LAs because it is the most novel and recent approach (Arnaoudova *et al*., 2013). Other possible anti-patterns detection techniques (e.g., inFusion, JDeodorant or PMD) can be used to confirm our findings. Another threats relate to our method for detecting post-release bugs. In effect, we have used a method that is widely applied in the literature (Kamei *et al*., 2013), (McIntosh *et al*., 2014), (McIntosh *et al*., 2016). Yet, we are aware that this accuracy is not perfect since it includes its authors' subjective understanding of the code smells (Moha *et al*., 2010). Additionally, DECOR accuracy may have an impact on our results since we may have classified a class without smells as a class involving smells and vice versa. In the future, we intend to apply other techniques and tools to confirm our findings (Moha *et al*., 2010).

Internal validity threats deal with alternative explanations of our results. It is important to mention that we do not claim causation, but we bring empirical evidence of the relationship between the presence of a particular family of smells and the occurrences of changes, and faults. Another threat is related to errors related to fault-fixing changes. We mitigated such a threat by not computing only the post-release defects, but also defects using the SZZ algorithm (Śliwerski *et al*., 2005).

Conclusion validity threats concern the relation between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used non-parametric tests, which do not make any assumption on the underlying distributions of the data, and, specifically, Fisher's exact test. Also, we based our conclusions not only on the presence of significant differences but also on the presence of a practically relevant difference, estimated by means of odds ratio measures. Last, but not least, we dealt with problems related to performing multiple Fisher tests using the Bonferroni correction procedure.

Reliability validity threats concern the possibility of replicating this study. We make publicly available all information and necessary details to replicate our study. Moreover, the source code repositories and issue tracking systems are publicly available to obtain the same data. The raw data used to compute the statistics presented in this paper are available online.

External validity threats concern the possibility of generalizing our results. We studied three systems having their corresponding control version system from where we extracted changes and fault fixes. It is true that three projects are not a large number. However, we analyzed a large number of releases, i.e., 30 releases in total. The investigated systems have different sizes and belong to different domains. Such a number of systems and releases may not be representative of all systems, and thus, we cannot guarantee that similar findings will be obtained when applying our approach to other open or closed source systems. Additionally, further validation on a larger set of systems from different domains is recommended to make sure our results are generalizable. Finally, we used a specific yet representative family of linguistic anti-patterns and design anti-

patterns. Different smells could be investigated in future work and could lead to different results.

5.5 Summary

This chapter provides further empirical evidence that design anti-patterns and linguistic anti-patterns relate to change- and fault-proneness. To study the relation between the detected families of design smells and change- and fault-proneness, we leveraged the change history of the studied systems using information from their Git/SVN versioning systems. We also mined their bug repositories. Interestingly, our findings show that LAs can make, in some cases, classes with DAPs more fault-prone when both occur in classes of object-oriented systems.

In addition, they indicate that, in a lot of cases, classes containing DAPs are more change- and fault-prone than classes with LAs. The occurrence of DAP in a class that experienced a LA has a strong relationship with change- and fault-proneness than the occurrence of LA in a class that experienced a DAP.

Based on our obtained results from Chapter 4 we found that, LAs hinder understandability of the code, but they do not eventually lead to have more changes and faults. Thus, development teams and quality assurance teams should better focus their refactoring efforts on components with design anti-patterns (while not neglecting linguistic anti-patterns) to assure good quality for their systems.

In the next Chapter, we further investigate the impact of LAs on the classes that have mutations between DAPs and DPs and their impact on change- and fault-proneness. Therefore, we first investigate the impact of design anti-patterns and design pattern evolution on change- and fault-proneness. Then, we investigate the impact that occurrences of LAs has on the change- and fault-proneness of evolving anti-patterns and design patterns.

CHAPTER 6 LINGUISTIC ANTI-PATTERNS, DESIGN ANTI-PATTERNS, DESIGN PATTERNS AND THEIR MUTATIONS AND FAULT-PRONENESS

The mutation of Design patterns and design anti-patterns has an impact on quality in terms of change- and fault-proneness. Linguistic anti-patterns have more negative impact on the mutated classes containing design (anti-patterns) than classes that only have design (anti-patterns).

In this chapter, we perform an empirical study to study the behaviors and impacts of linguistic anti-patterns, design anti-patterns, and design patterns on software quality in terms of Change- and Fault- Proneness.

6.1 Context

During software evolution, inexperienced developers may introduce design anti-patterns and–or linguistic anti-patterns when they modify software systems to fix bugs or to add new functionalities based on changes in requirements. Developers may also use design patterns to ensure software quality or as a possible cure for some design anti-patterns. Thus, during software evolution, design patterns and design anti-patterns are introduced, removed, and mutated from ones to the others by developers. Many studies investigated the evolution of design patterns and design anti-patterns and their impact on software development. However, existing studies investigated design patterns or design anti-patterns in isolation and did not consider their mutations and the impact of these mutations on software quality characteristics, such as fault-proneness. Linguistic anti-patterns affect code quality and negatively impact program comprehension. This, in turn, may lead to inconsistencies or faults during software change. Thus, the existence of linguistic anti-patterns may also contribute to the change- and fault-proneness of design patterns and design anti-patterns.

6.1.1 Research Problem and Contribution

Understanding the evolution of design anti-patterns and design patterns along with their relationships is important for software maintenance. Jaafar *et al*. (2013, 2014) investigated the relationships between design anti-patterns and design patterns in releases of several systems independently, one release at a time. They studied the relationships between classes playing roles in design patterns and design anti-patterns. They found that there are static relationships between design patterns and

design anti-patterns but that these relationships are temporary. They also showed that classes containing design anti-patterns, which have such relationships with design patterns, are more change-prone than other classes containing design anti-patterns, but are less fault-prone than the other design anti-pattern classes.

With software evolution, design patterns, design anti-patterns, and linguistic anti-patterns also evolve. Previous studies (Khomh et Guéhéneuc, 2008) have shown that the design of systems degrades over time, presumably due to the removal (or lack of use) of design patterns and the introduction of design anti-patterns. Understanding the dynamics behind the evolution of design patterns and design anti-patterns, in particular their mutations into one another, could help developers better prioritize maintenance activities and resource allocation. Besides, studying the impact of linguistic anti-patterns and their relationships with design anti-patterns and design patterns on change- and fault-proneness could also help developers to improve software quality.

This study is a quasi-replication of a previous study by (Jaafar *et al.*, 2014). Some of the objectives of this chapter are similar to those in the previous study by (Jaafar *et al.*, 2014), which are listed as follows:

- Calculate the probability of design anti-patterns mutations using Markov models.
- Compare classes with and without design anti-patterns to investigate the impact of the presence of design anti-patterns on fault-proneness.

However, we studied the systems at the commit level while Jaafar *et al.* (2014) investigations were at the release level. Besides, we consider both design patterns and design anti-pattern mutations during the evolution. In particular, we examine the impacts of design patterns and design anti-patterns mutations on change- and fault-proneness. In addition, we investigate the impacts of linguistic design anti-patterns on the change- and fault-proneness of the classes associated with mutating design patterns and–or design anti-patterns. We investigate seven open-source Java software systems to achieve the following six research investigations:

1. We investigate how design patterns and/or design anti-patterns mutate over time by modeling the behavior of these mutations using Markov models.
2. We investigate the most frequent types of changes that occur during design anti-patterns mutations.
3. We examine the impacts of these mutations on fault-proneness.
4. We study the types of changes to the code that lead to design patterns and design anti-patterns mutations.

5. We study the most fault-prone transitions from design patterns and–or design anti-patterns.

6. We study the impact of the co-existence of linguistic anti-patterns and design patterns or design anti-patterns on change- and fault- proneness.

We build one Markov chain ((Meyn et Tweedie, 2012)) for each system to model the mutations of design patterns and/or design anti-patterns. In such models, design anti-patterns and design patterns are nodes of the model, and the probability of the mutation between each pattern labels the arcs or edges between nodes. We compute the probability values by analyzing a set of snapshots of the systems.

We focus on thirteen design anti-patterns from (Brown *et al*., 1998) and six design patterns (Gamma, 1995) to investigate their relations with fault-proneness. For the detection of design anti-patterns and design patterns, we use DECOR by (Moha *et al*., 2010) and DeMIMA by (Guéhéneuc et Antoniol, 2008). We investigate seven different open-source systems from different sizes and application domains, Apache Ignite[1], Apache Solr [2], Eclipse IDE [3], Matsim [4], Mule [5], Nuxeo [6], and Ovirt [7]. We first detect the occurrences of design anti-patterns and design patterns in all the extracted snapshots of the systems and then investigate the types of mutations: persistent, deleted, introduced, and changed between these snapshots. Second, we build Markov models to compute the probability values of such mutations. The mutation probabilities allow us to understand how likely design patterns and/or design anti-patterns mutate into each other. Then, we use the SZZ algorithm by (Śliwerski *et al*., 2005) to find fault-inducing commits and investigate the impact of design patterns and–or design anti-patterns mutations on the fault-proneness of classes. In addition, we define thirteen types of change and examine them to discover what kinds of changes lead to mutations from design patterns and–or design anti-patterns. We also study the effects of such changes on fault-proneness.

6.1.2 Research Questions

As we mentioned before, this study is a quasi-replication of a previous work by (Jaafar *et al*., 2014). Addition to that work, we also consider design patterns mutations during the evolution. In particular, we examine the impacts of design patterns and design anti-patterns mutation on the

[1] `https://ignite.apache.org/`
[2] `http://lucene.apache.org/solr/`
[3] `https://www.eclipse.org/`
[4] `https://matsim.org/`
[5] `http://www.mulesoft.org/`
[6] `https://www.nuxeo.com/`
[7] `https://www.ovirt.org/`

change- and fault-proneness of classes. In addition, we investigate the impacts of linguistic anti-patterns on the change- and fault-proneness of classes associated with mutating design patterns and–or design anti-patterns. We investigate seven open-source Java software systems to answer the following five research questions:

- **RQ6.1:** ***Do design patterns and–or design anti-patterns mutate during the evolution of software systems? What is the probability of occurrence of different types of mutations?***

  We build Markov models showing which DPs and–or DAPs mutate into one another during a studied period of evolution. We consider both appearance and disappearance of DPs and DAPs occurrences. We observe that both DPs and DAPs mutate in the systems. We compute the probabilities of all possible mutations using the Markov models. We also present the most frequently mutated DPs and DAPs along the following four mutation types:

  1. Design Anti-patterns to Design Anti-patterns,
  2. Design Anti-patterns to Design patterns,
  3. Design patterns to Design Anti-patterns,
  4. Design patterns to Design patterns.

- **RQ6.2:** ***What types of changes lead to a mutation between design patterns and-or design anti-patterns?***

  DPs and design DAPs evolve through different types of changes as the system evolves. We define thirteen change types and investigate classes experiencing these change types and participating in DPs and–or DAPs. We see that different types of changes occur during the evolution of software systems and lead to different mutations. We study the impact of the types of changes on mutations between DPs and–or DAPs. We present the most prevalent type of changes leading to mutations from DAPs to DPs and vice-versa.

- **RQ6.3:** ***What is the fault-proneness of mutated design patterns and anti- patterns?What transitions lead to more fault-prone mutations?***

  DPs and DAPs may frequently mutate in other types of patterns during the evolution process. We study whether such mutations are risky regarding fault-proneness. We also present the riskiest transitions among DPs and DAPs. We observe that classes participating in mutated DAPs are more fault-prone than classes involved in mutated DPs. We also see that mutations from DAPs to DPs are more faulty than the other mutations.

- **RQ6.4:** ***Do specific types of changes lead to increase fault-proneness during design patterns and-or design anti-patterns mutations?***

We investigate whether the types of changes impact fault-proneness. We examine faulty-classes and check whether a mutation occurred during the evolution of these classes. We also examine all changes experienced by the classes during the evolution of the systems. We observe that some of the change types make the systems more fault-prone. We study whether specific types of changes cause the mutations of DPs to DAPs and vice-versa more fault-prone.

- **RQ6.5:** ***Do the occurrences of Linguistic anti-patterns increase change- and fault-proneness during design patterns and-or design anti-patterns mutations?***

  Design patterns, design anti-patterns, and linguistic anti-patterns may co-exist. However, although important, we do not know whether and how their co-existence impact software quality by increasing change- and–or fault-proneness. Thus, we study whether the existence of linguistic anti-patterns impact change- and fault-proneness of the classes associated with evolving DPs and DAPs. Our results show that the co-existence of linguistic anti-patterns and design patterns or design anti-patterns leads to increased change- and fault-proneness in the mutated classes during the software evolution.

The results of these five research questions show that there is a high probability for some design patterns and–or design anti-patterns to mutate to others types of design patterns and–or design anti-patterns. The changes that lead to the mutations are mostly structural changes, in particular the addition of large number of attributes or long methods. Results also show that patterns mutations can increase the fault-proneness of a system. Also, the co-existence of the linguistic anti-patterns and design patterns or design anti-patterns increase the change- and fault-proneness of the concerned classes.

6.2 Study Design

We use the methodology presents in Figure 6.1 to answer our research questions. We first extract source code of the revisions at 500 commit-intervals from the studied systems' Git repositories. Then, we detect design anti-patterns and design patterns for each of the selected snapshots of each system. We then create Markov models based on the detected design anti-patterns and design patterns to analyze their behaviors during evolution. Afterwards, we identify change types and faulty classes throughout the period of evolution that we analyzed. We study all the change types that lead to fault(s) during evolution. We also detect linguistic anti-patterns in all the systems to investigate their impacts on change- and fault-proneness. We explain each step in details in the following sections.

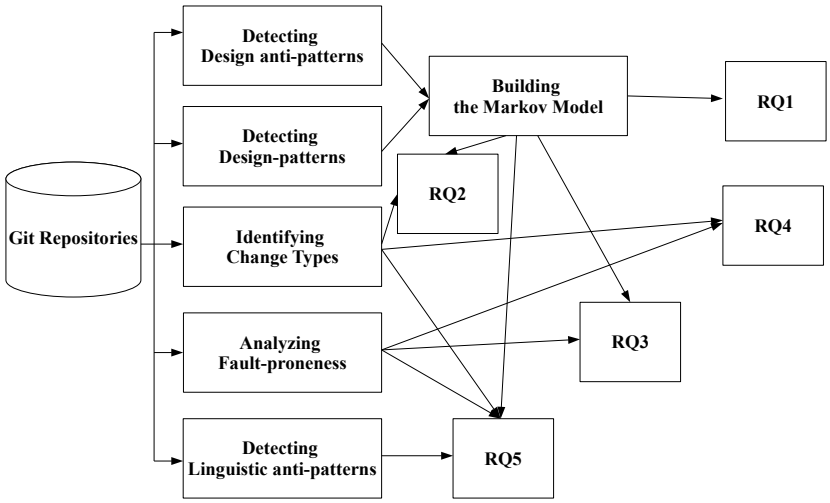

Figure 6.1 Schematic diagram of the methodological steps of the study presented in this chapter.

6.2.1 Studied Design Anti-patterns, linguistic Anti-patterns and Design patterns

We select thirteen design anti-patterns in this chapter. These design anti-patterns introduced by (Brown *et al*., 1998) expresses problems with data, complexity, size, and the features related to classes. They have also been studied in previous work (Khomh *et al*., 2012). We summarized their definitions in chapter 2, Section2.2.1. More details about these anti-patterns are available elsewhere (Romano *et al*., 2012). We also studied eight design patterns from (Gamma, 1995), described in chapter 2, Section 2.3.1. We also studied all the types of linguistic anti-patterns described in Chapter 2, Section 2.1.1.

6.2.2 Studied Systems

We consider seven Java-based open-source systems for our study as shown in table Table 6.1. We select these systems based on diversity in code base size, application domains, and their evolution history. These systems are comparatively bigger with many classes and mature enough to have longer evolution histories with a large number of commits. As these systems have evolved over the years and they have many versions to provide a rich dataset for analyzing the change- and fault-proneness of the evolved classes. Table 6.1 provides descriptive statistics about the selected systems.

Eclipse IDE for Java[8] is an IDE for any Java developer. The IDE use the Java development Tools (JDT) to develop Java applications. It contains CVS, and Git client. It also includes XML Editor, Mylyn as a task management system, build supports for Maven and WindowBuilder.

[8] `https://www.eclipse.org/`

Table 6.1 Subject systems analyzed

System	Applicaion domain	# Commits	LOC	Issue Tracker
Eclipse for Java	IDE	281,396	9,064,794	Bugzilla
Nuxeo Platform	Colaboration management	265,380	5,741,131	Jira
oVirt	Visualization platform	149,128	2,764,655	Bugzilla
Matsim	Transportation management	44,200	1,602,877	Atlassian
Apache Solr	Search server	30,995	658,711	Jira
Apache Ignite	Distributed DB platform	24,104	1,471,036	Jira
Mule Community Edition	Integration platform	22,891	309,616	Jira

Nuxeo[9] also called Nuxeo Platform, is an open-source context management and collaboration platform, which provides different information management solutions for developers to build business applications.

oVirt[10] is a visualization management platform written in Java. The platform provides a centralized management of resources, storage, and virtual machines, which allows to manage your entire enterprise infrastructure. oVirt contains two main parts including oVirt engine and oVirt node.

Matism[11] is a useful framework to build large-scale transport solutions. This application has been used in different areas of transportation. The development team provides a comprehensive documentation for users and developers to ease the usage and maintainability of the system.

Apache Solr from the Apache Lucene project[12] is an open-source Java search server. It searches websites, databases, and files. Solr is a popular and fast search platform which uses the Lucene Java search library at its core for full-text indexing and search. It runs as a standalone full-text search server.

Apache Ignite[13] is a powerful in-memory computing platform used as database and caching system. It has several features: it has been used to solve complex problems pertaining to speed and scalability; it also can be used as in-memory data grid and caching capabilities to accelerate existing relational and NoSQL databases; and as a distributed SQL to gain horizontal scalability, strong consistency, and high availability.

Mule[14] is a runtime engine of a Java-based enterprise service bus (ESB) and integration platform. Developers can connect applications together quickly and easily, and enable them to exchange data. Mule has several features including service creation and hosting, service mediation, message

[9] https://www.nuxeo.com/
[10] https://www.ovirt.org/
[11] https://matsim.org/
[12] http://lucene.apache.org/solr/
[13] https://ignite.apache.org/
[14] http://www.mulesoft.org/

routing, and data transformation.

6.2.3 Building a Mutation Model

We build a Markov model ((Meyn et Tweedie, 2012)) for each system to show the mutations of design anti-patterns and design patterns during evolution. For a clear understanding of these mutations, we present the mutations as directed graphs. We consider each pattern across all the selected snapshots of a systems as a node of the Markov model. A Markov model indicates a set of all possible mutations for that pattern during the evolution of the system. We consider a sequence of random variables *e.g.,*$X_1, X_2, X_3, ..., X_n$ for each pattern. The possible values of X_i form a countable set S called the state space of the model.

For example, Figure 6.2 presents a directed graph where the nodes are design anti-patterns and design patterns while the edges represent mutations from one node (pattern or design anti-pattern) to another. The edges are labeled with the probabilities of the mutation from source patterns to target patterns. This Markov model shows the mutations of the Builder design pattern across the selected snapshots of Matsim. The sum of all the probabilities of mutations from each design patterns and–or design anti-patterns to other patterns in a Markov model is equal to one. For each pattern, we have a graph with the computed probabilities for all possible mutations from this pattern to the other patterns during the evolution of each system.

6.2.4 Analyzing Fault-proneness

We use the SZZ algorithm by (Śliwerski *et al.*, 2005) to identify commits that introduce faults, referred to as *fault-inducing commits*. All the studied systems are version-controlled by Git. For each system, we first apply heuristics by (Fischer *et al.*, 2003) to link commits to bugs. We use regular expressions to detect bug-IDs from the systems commit-logs. Developers of different projects may

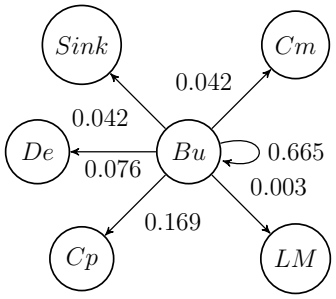

Figure 6.2 Builder (Bu) mutation among the different revisions of Matsim.

use different patterns to write bug-IDs in the commit logs. To ensure the accuracy of the results, we tune our regular expressions on our dataset incrementally through manual analysis.

For an identified bug B, we extract files that fixed the bug (bug-fixing files) by using the `git diff` command. Next, we retrieve the modified and deleted lines from the bug-fixing files. The SZZ algorithm assumes that prior commits that modified the lines (prior to the opening of the bug report), which are modified or deleted by the bug-fixing files, are fault-inducing commits. To identify such prior commits, on each bug-fixing files, we apply the `git blame` command to retrieve a list of previous commits that modified these files. From the result list, we only keep the commits that modified the lines, which are then changed by corresponding bug-fixing commits. We filter out the commits whose submission date is later than its related bug creation date. The remaining commits are considered as the fault-inducing commits for the bug B. For a given bug B, the SZZ algorithm will yield a list of commit IDs and the fault-inducing files in the commits related to the bug B. We use regular expressions to map the fault-inducing files to our studied classes.

6.2.5 Identifying Change Types

Different types of changes can affect software systems with different impacts on fault-proneness. For example, changes to comments are less likely to lead to faults than changes to method invocations. Table 6.2 shows the change types considered in this study and which were also used in previous work by (An et Khomh, 2015).

We use srcML ((srcml, 2016)) to transform each file of the systems into an XML document, in which each code element is tagged by its type or function, *e.g.,*a class declaration, a parameter list, or a control flow statement. We use a Python script to automatically compare the srcML tags between each two subsequent snapshots and extract their differences. The removed tags from the old snapshot and the added tags in the new snapshot are referred as *changed tags*. We manually group the unique changed tags into a series of *change types*. Table 6.2 shows the change types and their corresponding srcML (changed) tags. We group some change types together, because each group of change types represents the same kind of code changes, which have similar impacts on the source code. Thus, changes in the same group may have a similar impact on the fault-proneness of the code.

For a given file F in a specific commit C, our approach yields a list of change types listed in Table 6.2. As we study design- and design anti-patterns at commit level, for each selected snapshots of system R, we aggregate the change types related to F in the commits $\{C_1, C_2, ..., C_n\}$, which belong to R.

Table 6.2 Change types identified from the source code of the systems studied

Change type	**srcML tag(s)**
Access	*super, public, private, protected, extern*
Class	*extends, class, interface, implements, class_decl*
Code block	*expr_stmt, expr, block*
Comment	*annonation, comment, @type, @format*
Control flow	*while, do, if, else, elseif, break, goto, for, foreach, control, continue, then, switch, case, return, incr, default, condition*
Declaration	*decl_stmt, modifier, specifier, decl, function_decl, literal, label, empty_stmt, construction_decl, annonation_dfn*
Exception	*assert, try, catch, throw, throws, finally*
Import	*import, package*
Invocation	*call*
Method	*constructor, default, static, type, lambda, function, function_decl, unit*
Operator	*index, synchronized, enum, operator, ternany*
Parameter	*argument, param, parameter_list, argument_list, parameter*
Renaming	*renaming, name*

6.3 Study Results

In this section, we present the findings of our study by answering the five research questions that we defined in Section 6.1.2.

6.3.1 RQ6.1: Do design patterns and–or design anti-patterns mutate during the evolution of software systems? What is the probability of occurrence of different types of mutations?

Motivation For software maintenance, a good understanding of the evolution of design patterns is important because it can help developers to identify and circumvent risky design patterns and prevent design anti-patterns from appearing ((Jaafar *et al.*, 2014)). While some tools can find software entities and their patterns of evolution automatically (e.g., (Van Emden et Moonen, 2002; Lanza et Marinescu, 2007; Rapu *et al.*, 2004; Vaucher *et al.*, 2009)), no previous work investigated the mutation of design patterns and design anti-patterns.

Computing probability values for all possible mutations We identify design anti-patterns using the detection tools described in Section 2.2.2. We apply the detection tools on the selected snapshots of each of the systems listed in Table 6.1. We select snapshots at every 500 commit interval. Our selection of commit interval period is adequate to detect changes occurring between two subsequent snapshots (according to (Hassan, 2009; Canfora *et al.*, 2010)). We automatically compare each two subsequent snapshots to compute the numbers of added or deleted occurrences of patterns. Using this information, we build one Markov model for each of the systems to compute the probability of mutations of design patterns and design anti-patterns.

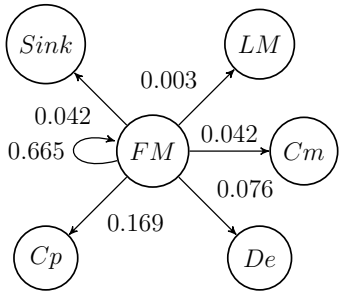

Figure 6.3 FactoryMethod (FM) mutation among the different revisions of Eclipse.

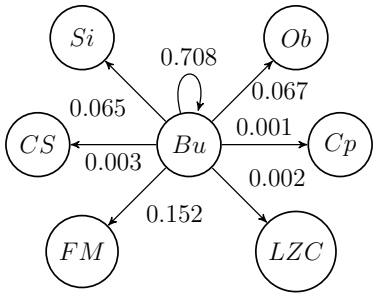

Figure 6.4 Builder (Bu) mutation among the different revisions of Nuxeo.

Tables 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, and 6.9 show all the mutations of design patterns and design

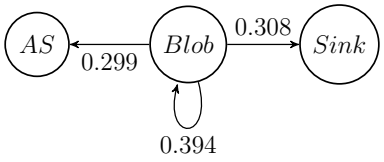

Figure 6.5 Blob (BL) mutation among the different revisions of oVirt.

Table 6.3 Change probabilities of design anti-patterns and design patterns in Eclipse IDE

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.013	**0.972**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.015
Bl	0.007	**0.27**	**0.447**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.277**
CS	0.001	0.000	0.001	**0.993**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.005
CC	0.000	0.000	0.000	0.012	**0.973**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.014
LC	0.000	0.000	0.000	0.000	**0.5**	**0.5**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.012	**0.859**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.006
LM	0.000	0.000	0.000	0.001	0.000	0.000	0.024	**0.95**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.025
LP	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.009	**0.966**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.025
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.273**	**0.455**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.273**
SC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.876**	0.000	0.000	0.000	0.000	0.049	0.000	0.002	0.05
Cm	0.000	0.010	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.668**	0.091	0.09	0.000	0.000	0.000	0.000	**0.141**
Cp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042	**0.169**	**0.665**	0.076	0.000	0.000	0.000	0.042
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.892**	**0.107**	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.020	**0.966**	0.014

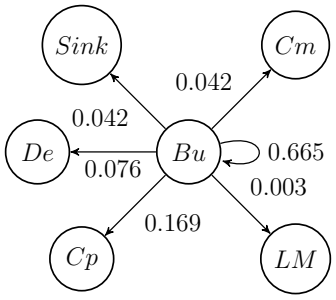

Figure 6.6 Builder (Bu) mutation among the different revisions of Matsim.

Table 6.4 Change probabilities of design anti-patterns and design patterns in Nuxeo

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.014	**0.969**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.017	0.000	0.000	0.000	0.000	0.000
Bl	0.003	**0.283**	**0.417**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.297**	0.000	0.000	0.000	0.000	0.000
CS	0.002	0.000	0.008	**0.982**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.008	0.000	0.000	0.000	0.000	0.000
CC	0.001	0.000	0.000	0.040	**0.912**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.047	0.000	0.000	0.000	0.000	0.000
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.048	**0.910**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042	0.000	0.000	0.000	0.000	0.000
LM	0.001	0.000	0.000	0.003	0.000	0.000	0.028	**0.937**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.030	0.000	0.000	0.000	0.000	0.000
LP	0.000	0.001	0.000	0.006	0.000	0.000	0.002	0.017	**0.946**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.000	0.000	0.000	0.000	0.000
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SC	0.012	0.000	0.000	0.012	0.000	0.000	0.000	0.000	0.000	0.000	0.067	**0.824**	0.000	0.000	0.000	0.000	0.000	0.85	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cm	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.019	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.133**	0.000	0.000	**0.105**	**0.760**	0.000

Table 6.5 Change probabilities of design anti-patterns and design patterns in oVirt

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.018	**0.971**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.011
Bl	0.000	**0.299**	**0.394**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.308**
CS	0.006	0.000	0.003	**0.982**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.009
CC	0.000	0.000	0.000	0.012	**0.975**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.013
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.001	**0.998**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001
LM	0.000	0.000	0.000	0.000	0.000	0.000	0.012	**0.977**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.011
LP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.008	**0.969**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.022
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.082	**0.857**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.061
SC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cm	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.019	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.097**	**0.798**	**0.103**

anti-patterns that occurred during the selected period of evolution, of the studied systems. In these tables, we add two additional states (source and sink) to cover those mutations from no patterns to design (anti-) patterns, and–or from design (anti-) patterns to no patterns referred to as source and sink respectively. *Source* indicates the new classes which did not participate in the occurrences of design patterns and–or design anti-patterns before the selected period of evolution. A *sink*, on the other hand, represents the classes which participated in occurrences of design patterns and–or design anti-patterns before the selected period of evolution but not after.

For example, SpaghettiCode has the most representative mutation probability from source in Mule (Table 6.9), and Blob and to Sink in Apache Solr (Table 6.7). Table 6.10 shows the most representative design patterns and design anti-patterns regarding the mutation probability values. Here, the most representative design (anti-) patterns are those having the highest probability value of

Table 6.6 Change probabilities of design anti-patterns and design patterns in Matsim

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.066	**0.893**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.040	0.000	0.000	0.000	0.000	0.000
Bl	0.003	**0.372**	**0.279**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.346**	0.000	0.000	0.000	0.000	0.000
CS	0.025	0.000	0.037	**0.9**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.038	0.000	0.000	0.000	0.000	0.000
CC	0.004	0.001	0.002	0.075	**0.848**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.069	0.000	0.000	0.000	0.000	0.000
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.003	0.000	0.000	0.000	**0.1**	**0.831**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.066	0.000	0.000	0.000	0.000	0.000
LM	0.003	0.001	0.002	0.013	0.000	0.000	0.063	**0.86**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.059	0.000	0.000	0.000	0.000	0.000
LP	0.003	0.000	0.003	0.013	0.000	0.000	0.007	0.034	**0.887**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.053	0.000	0.000	0.000	0.000	0.000
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.021	**0.596**	0.000	0.000	0.000	0.000	0.000	0.000	**0.193**	0.000	0.000	0.000	0.000	0.000
SC	0.026	0.000	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.038	**0.834**	0.000	0.000	0.000	0.000	0.000	**0.099**	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.091	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.182**	**0.455**	0.000	0.000	0.000	0.000	**0.273**	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.001	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.546**	0.000	0.001	**0.272**	0.000	**0.141**	0.000	0.030	0.000
Cm	0.000	0.000	0.000	0.001	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.030	0.000	**0.371**	0.075	**0.387**	**0.117**	0.012	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.002	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.037	0.044	**0.850**	0.061	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.001	0.000	0.000	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.036	0.074	**0.825**	0.059	0.000	0.000	0.000	0.000
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.049	**0.912**	0.034	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.000	0.000	0.009	**0.962**	0.000

Table 6.7 Change probabilities of design anti-patterns and design patterns in ApacheSolr

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bl	0.000	**0.321**	**0.365**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.313**
CS	0.000	0.000	0.009	**0.983**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.008
CC	0.000	0.000	0.000	0.033	**0.93**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.037
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.072	**0.859**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.069
LM	0.000	0.000	0.001	0.002	0.000	0.000	0.033	**0.928**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.036
LP	0.000	0.001	0.000	0.008	0.000	0.000	0.007	0.079	**0.779**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.125
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.088	**0.84**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.072
SC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.013	0.000	**0.981**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.006
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.963**	0.000	0.000	0.000	0.000	0.005	0.000	0.000	0.027
Cm	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.012	**0.874**	0.005	0.000	0.000	0.000	0.092
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.996**	0.000	0.000	0.000	0.004
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.007	**0.981**	0.012

mutation from one type to another type of design (anti-) pattern.

Analyzing pattern evolution We observe that not all design patterns and design anti-pattern occurrences undergo changes; some of them remain stable during evolution. For example, “Lazy-Class” and ‘MessageChain” are instances of stable design anti-patterns, while “Prototype” is a persistent design pattern in the Apache Solr system. In Matsim, design anti-patterns “SwissArmyKnife”, “LazyClass”, and ‘MessageChain” and design patterns “Observer” and “ProtoType” were stable.

However, in general, design anti-patterns tend to evolve in all studied systems. We recognized that more than half of design anti-pattern occurrences mutated to another design patterns or design anti-patterns across different snapshots of each system. For example, in Matsim (Table 6.6), 86% (probability value 0.86) “LongMethod” occurrences remain stable in the system and with a

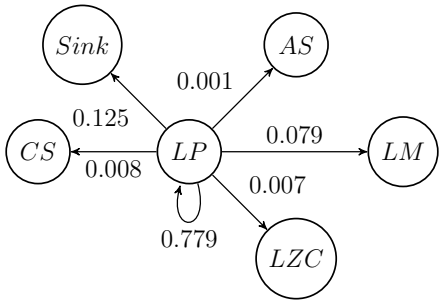

Figure 6.7 LongParameterList (LP) mutation among the different revisions of ApacheSolr.

Table 6.8 Change probabilities of design anti-patterns and design patterns in ApacheIgnite

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bl	0.000	**0.375**	**0.33**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.295**
CS	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
CC	0.000	0.000	0.000	0.053	**0.905**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.015	**0.97**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.015
LM	0.000	0.000	0.000	0.004	0.000	0.000	0.045	**0.905**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.047
LP	0.000	0.000	0.000	0.009	0.000	0.000	0.006	0.051	**0.851**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.081
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.976**	0.000	0.000	0.000	0.000	0.004	0.000	0.000	0.018
Cm	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000
De	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.003	**0.994**	0.003

probability of 14%, they are mutated to other forms of design patterns and–or design anti-patterns. Similarly, in Ovirt (Table 6.5), "Blob" with the 39.4% probability remain persistent in the system, while 29.9% of them mutated to "AntiSingleton", and with a probability of 30.8%, they are mutated to other patterns. As the last example, in Eclipse, 45.5% of "RefusedParentBequest " occurrences retain their states from a release to another; 27.3% are mutated to "MessageChain", and 27.3% are mutated to other forms of design patterns and–or design anti-patterns. We saw fewer changes among design patterns. As an example, in Apache Ignite, 97.6% of "Command" occurrences remained stable, and only 0.4% are changed to another type of design anti-patterns or design patterns.

For a better understanding of the design pattern and design anti-pattern mutations, Figures 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, and 6.9 present the Markov models as graphs with the most representative mutations and with the highest probabilities (weights on edges) for each of the systems. Since drawing all possible probabilities make the graphs complex, we choose a threshold for each system to reduce the number of edges in each graph. The chosen threshold is 0.100 for all the studied systems. The gray cells in Tables 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, and 6.9 represents all probabilities

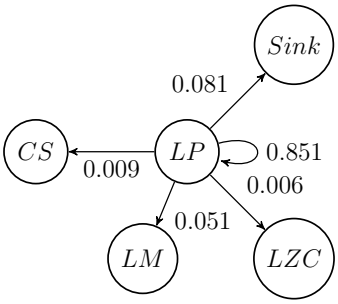

Figure 6.8 LongParameterList (LP) mutation among the different revisions of ApacheIgnite.

Table 6.9 Change probabilities of design anti-patterns and design patterns in Mule

	Source	AS	Bl	CS	CC	LC	LZC	LM	LP	MCh	RP	SC	SG	SA	Bu	Cm	Cp	FM	De	Ob	PT	Si	Sink
AS	0.032	**0.937**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.030	0.000	0.000	0.000	0.000	0.000
Bl	0.000	**0.313**	**0.313**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.374**	0.000	0.000	0.000	0.000	0.000
CS	0.003	0.000	0.006	**0.963**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.000	0.000	0.000	0.000	0.000
CC	0.001	0.000	0.000	0.071	**0.843**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.084	0.000	0.000	0.000	0.000	0.000
LC	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
LZC	0.000	0.000	0.000	0.000	0.000	0.030	**0.946**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.024	0.000	0.000	0.000	0.000	0.000
LM	0.000	0.000	0.000	0.010	0.000	0.000	0.039	**0.907**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.044	0.000	0.000	0.000	0.000	0.000
LP	0.000	0.000	0.000	0.014	0.000	0.000	0.009	**0.115**	**0.676**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.186**	0.000	0.000	0.000	0.000	0.000
MCh	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RP	0.000	0.000	0.000	0.033	0.000	0.000	0.067	0.000	0.000	**0.233**	**0.233**	0.000	0.000	0.000	0.000	0.000	0.000	**0.433**	0.000	0.000	0.000	0.000	0.000
SC	**0.096**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.074	**0.649**	0.000	0.000	0.000	0.000	0.000	**0.181**	0.000	0.000	0.000	0.000	0.000
SG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.053	**0.947**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SA	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bu	0.000	0.000	0.000	0.003	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.708**	0.000	0.001	**0.152**	0.000	0.067	0.000	0.065	0.000
Cm	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.019	0.000	**0.687**	**0.096**	**0.193**	0.000	0.000	0.000	0.000	0.000
Cp	0.000	0.000	0.000	0.002	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.058	**0.882**	0.054	0.000	0.000	0.000	0.000	0.000
FM	0.000	0.000	0.000	0.003	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042	0.087	**0.764**	**0.099**	0.000	0.000	0.000	0.000
De	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.014	**0.971**	0.013	0.000	0.000	0.000
Ob	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.024	0.024	**0.951**	0.000	0.000	0.000
PT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1**	0.000	0.000
Si	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.029	0.000	0.000	0.006	**0.964**	0.000

that are displayed in these figures. We used those figures to present which types of design (anti-) patterns are usually mutated over the software evolution. This information can help developers to focus on such types to avoid the mutations that can have negative impacts on the quality of the systems.

Our results show that design patterns and design anti-patterns mutate during the evolution of software systems. In most of the studied systems, more than half of the design anti-pattern occurrences mutated during the evolution. On the other hand, in most of the systems, almost all the design pattern occurrences remained stable during the evolution process. We also observe that Blob and Command are the design anti-patterns and design patterns, which have higher mutation probabilities.

Table 6.10 Most representative design pattern and design anti-patterns mutations with mutation probabilities

System	Mutation Type	From	To	Probability
Apache Ignite	Design Anti-pattern → Design Anti-pattern	Blob (Bl)	AntiSingleton (AS)	0.375
	Design Anti-pattern → Design pattern	-	-	-
	Design pattern → Design Anti-pattern	-	-	-
	Design pattern → Design pattern	Builder (Bu)	Observer (ob)	0.004
Apache Solr	Design Anti-pattern → Design Anti-pattern	Blob (Bl)	AntiSingleton (AS)	0.321
	Design Anti-pattern → Design pattern	-	-	-
	Design pattern → Design Anti-pattern	-	-	-
	Design pattern → Design pattern	FactoryMethod (FM)	Composite (Cp)	0.012
Eclipe IDE	Design Anti-pattern → Design Anti-pattern	LargeClass (LC)	ComplexClass (Cc)	0.500
	Design Anti-pattern → Design pattern	-	-	-
	Design pattern → Design Anti-pattern	FactoryMethod (FM)	LongMethod (LM)	0.003
	Design pattern → Design pattern	FactoryMethod (FM)	Composite (Cp)	0.169
Matsim	Design Anti-pattern → Design Anti-pattern	Blob (Bl)	AntiSingleton (AS)	0.372
	Design Anti-pattern → Design pattern	Blob (Bl)	FactoryMethod (FM)	0.346
	Design pattern → Design Anti-pattern	Command (Cm)	SwissArmyKnife (SA)	0.030
	Design pattern → Design pattern	Command (Cm)	FactoryMethod (FM)	0.387
Mule	Design Anti-pattern → Design Anti-pattern	Blob (bl)	AntiSingleton(AS)	0.313
	Design Anti-pattern → Design pattern	RefusedParentBequest (RP)	FactoryMethod (FM)	0.433
	Design pattern → Design Anti-pattern	Command (Cm)	SwissArmyKnife (SA)	0.019
	Design pattern → Design pattern	Command (Cm)	FactoryMethod	0.193
Nuxeo	Design Anti-pattern → Design Anti-pattern	Blob (bl)	AntiSingleton(AS)	0.283
	Design Anti-pattern → Design pattern	Blob (Bl)	FactoryMethod (FM)	0.297
	Design pattern → Design Anti-pattern	Singleton (Si)	LazyClass (ZC)	0.004
	Design pattern → Design pattern	Singleton (Si)	FactoryMethod (FM)	0.133
Ovirt	Design Anti-pattern → Design Anti-pattern	Blob (bl)	AntiSingleton(AS)	0.299
	Design Anti-pattern → Design pattern	-	-	-
	Design pattern → Design Anti-pattern	Singleton (Si)	AntiSingleton (AS)	0.001
	Design pattern → Design pattern	Singleton (Si)	Prototype (PT)	0.097

6.3.2 RQ6.2: What types of changes lead to a mutation between design patterns and-or design anti-patterns?

Motivation Understanding the causes of design patterns and–or design anti-patterns mutations is a key knowledge during software maintenance. Thus, studying the type of changes related to design patterns and design anti-patterns mutation could help developers to focus on the most frequent change types triggering patterns mutations. In this research question, we identify the types of changes to understand why and how design patterns and–or design anti-patterns mutate between two successive snapshots.

Analyzing change types We use srcML[15] to create an XML file for each system and match with the scrML tags to find changed tags as we explained in Section 6.2.5. Then, we categorize change types based on our defined categories (in Table 6.2) and compare the percentages of each change type for each of the systems. We apply the same methodology for all the subject systems. Fig. 6.14

[15]https://www.srcml.org/

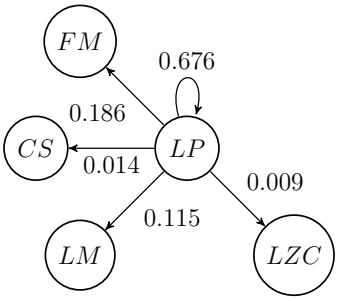

Figure 6.9 LongParameterList (LP) mutation among the different revisions of Mule.

shows the types of changes in Apache Ignite classes participating in design anti-patterns and design patterns, respectively. Table 6.11 also presents the numbers of each change types in this system. In Apache Ignite, we observe that “Access”, and “Renaming” are the least and most representative change types for both design patterns and design anti-patterns, which lead to many changes in occurrences of both design anti-patterns and design patterns.

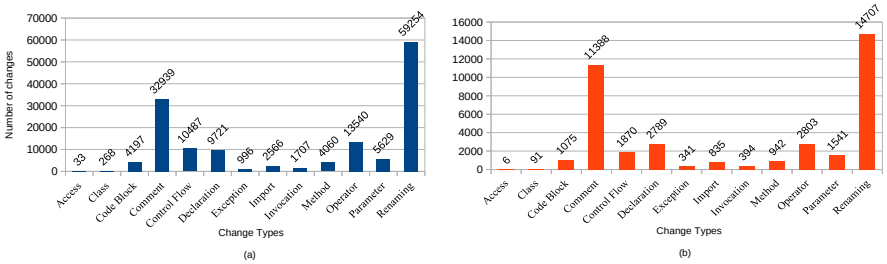

Figure 6.10 *Number of different types of changes in Eclipse classes with (a) design anti-patterns and (b) design patterns.*

Analyzing change types for mutations During evolution, occurrences of design patterns and design anti-patterns can also mutate into other design patterns and design anti-patterns. To analyze change types related to pattern mutation for this research question (RQ6.2), we investigate which types of changes lead to such mutations. The results of detected change types are contained in CSV files related to changed classes participating in design patterns and design anti-patterns for each system. Each CSV file includes the name of the two subsequent snapshots compared for

Table 6.11 Number of different types of changes in design patterns and design anti-patterns

Systems →	**Eclipse**		**Nuxeo**		**oVirt**		**Matsim**		**Apache Solr**		**Apache Ignite**		**Mule**	
Change Types ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	AP	**DP**	**AP**	**DP**	**AP**	**DP**	**AP**	**DP**	**AP**	**DP**	**AP**	**DP**	**AP**	**DP**
Access	33	6	85	0	174	9	271	117	34	15	11	55	20	12
Class	268	91	236	6	3804	90	1697	690	721	155	781	218	443	198
Code block	4197	1075	1070	13	13923	233	8805	3203	3873	459	3903	203	1430	413
Comment	32939	11388	9269	109	15013	411	22519	9287	9298	2616	14554	1567	6354	3780
Control Flow	10487	1870	966	10	6440	118	5398	2094	3166	636	3433	133	916	384
Declaration	9721	2789	3133	24	27605	487	24244	9609	8803	1392	9527	349	3904	1103
Exception	996	341	946	1	619	29	1602	314	1696	457	2076	64	505	173
Import	2566	835	2734	23	18819	211	13013	4679	4024	491	4584	394	3234	793
Invocation	1707	394	556	4	7312	91	8520	3026	2598	287	2069	75	945	240
Method	4060	942	1487	29	13792	292	4702	1922	3215	511	3364	266	1940	747
Operator	13540	2803	3533	8	35513	403	57112	24326	7963	702	7207	525	5241	1975
Parameter	5629	1541	2179	3	24488	292	22080	5149	8375	1069	9024	332	3252	756
Renaming	59254	14707	16259	23	262491	3536	294661	145720	44422	4811	63961	4396	28110	9738
# Changed classes	10957	3155	5402	81	34780	55	32596	13768	7956	1192	9290	857	5684	2175
Total classes	20331	7574	39051	1263	142537	2482	62272	79480	32332	5490	27080	5796	47146	17553

AP, DP= Number of changes in design anti-patterns and design patterns respectively

Table 6.12 Number of different types of changes in design patterns and design anti-patterns mutation.

Systems→	**Eclipse**		**Nuxeo**		**oVirt**		**Matsim**		**Apache Solr**		**Apache Ignite**		**Mule**	
Change Types ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	APDP	**DPAP**	**APDP**	**DPAP**	**APDP**	**DPAP**	**APDP**	**DPAP**	**APDP**	**DPAP**	**APDP**	**DPAP**	**APDP**	**DPAP**
Access	0	0	0	1	1	1	0	12	3	0	0	0	0	1
Class	3	1	2	0	10	0	4	29	1	7	2	4	6	3
Code block	1	2	1	4	21	22	10	132	4	1	1	3	27	17
Comment	102	192	7	4	69	31	38	739	129	242	58	23	160	85
Control Flow	28	38	0	3	7	6	12	96	44	31	16	4	21	3
Declaration	56	78	3	2	82	38	56	412	141	90	33	16	49	32
Exception	4	6	0	0	4	1	6	28	19	13	14	4	3	2
Import	20	14	3	2	32	18	28	257	60	34	17	16	28	18
Invocation	6	7	0	0	15	20	7	183	1	6	5	6	10	3
Method	16	18	2	2	41	24	7	96	64	43	10	12	23	13
Operator	38	37	2	0	43	66	95	523	22	20	6	13	87	32
Parameter	22	41	0	0	37	20	16	423	25	47	22	29	13	22
Renaming	675	469	3	2	297	1036	462	5096	165	406	100	63	334	280
# Changed classes	71	77	7	6	61	54	58	681	58	66	34	29	53	39

APDP, DPAP= Number of changes in design anti-patterns to design patterns and design patterns to design anti-patterns mutations respectively

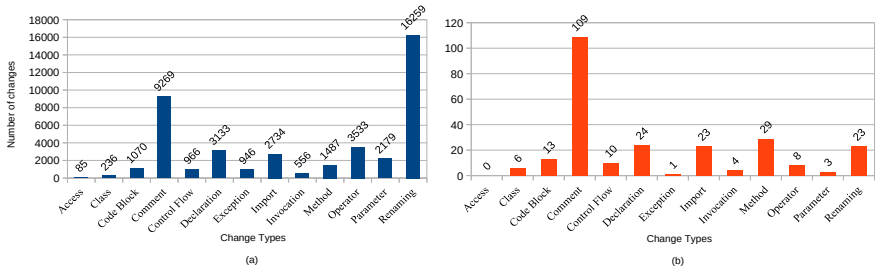

Figure 6.11 *Number of different types of changes in Nuxeo classes with (a) design anti-patterns and (b) design patterns.*

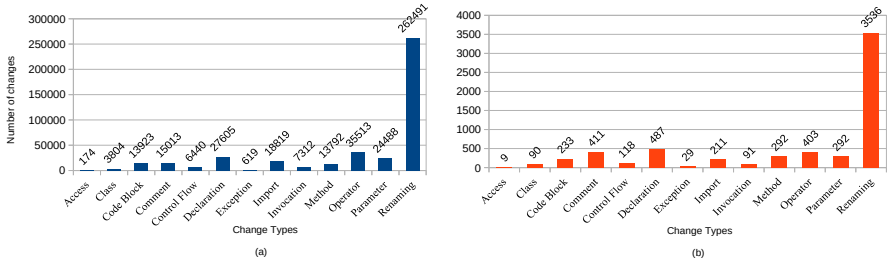

Figure 6.12 *Number of different types of changes in oVirt classes with (a) design anti-patterns and (b) design patterns.*

change detection, the type of changes and the names of the classes changed. We compare two CSV files related to design patterns and design anti-patterns. By comparing the names of classes participating in occurrences of design patterns and design anti-patterns, we find the same class names which indicate that we have a mutation from design anti-patterns to design patterns and vice versa. Tables 6.12 shows the number of each change types during the mutation for all the studied systems.

Results show that, in Apache Ignite, “Renaming”, “Comment”, and “Declaration” changes have the most impact on DAPs to DPs mutations. It is almost the same for the DPs to DAPs mutations but “Parameter” has more impact than “Declaration”. In Apache Solr, and Eclipse for both DAPs to DPs and DPs to DAPs mutations, “Renaming”, “Declaration”, and “Comment” are the most representative change types. In Matsim, “Renaming”, “Operator”, and “Declaration” have the most impact on DAPs to DPs mutations, while for the reverse mutations (i.e., DPs to DAPs), “Renaming”, “Comment”, and “Operator” have the most impact. In Mule, in both DAPs to DPs and DPs to DAPs mutations, “Renaming”, “Comment”, and “Operator” are the most representative change types. In Nuxeo, there are few mutations, in which “Comment”, “Renaming”, and “Declaration” are the most representative change types in DAPs to DPs mutations, and for the vice versa, we find “Comment”, “Code Block”, and “Control Flow” as the most DPs to DAPs mutation

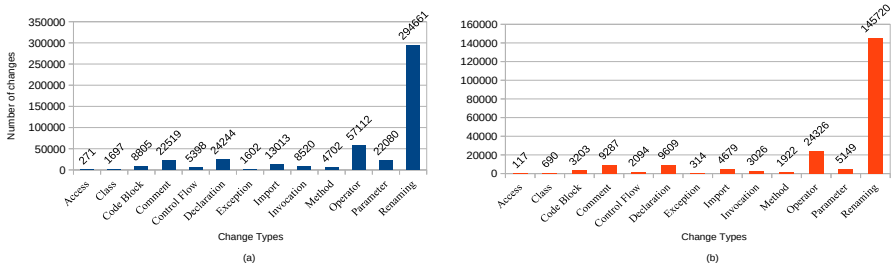

Figure 6.13 *Number of different types of changes in Matsim classes with (a) design anti-patterns and (b) design patterns.*

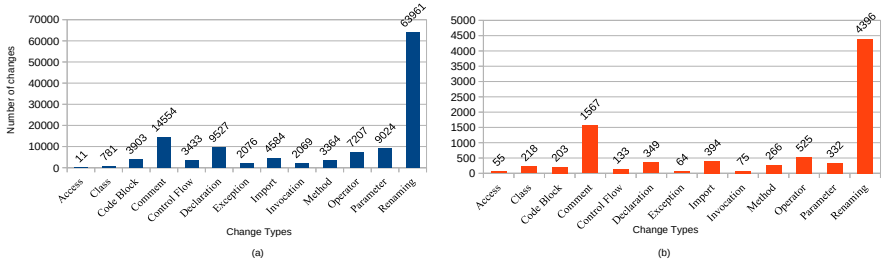

Figure 6.14 *Number of different types of changes in Apache Ignite classes with (a) design anti-patterns and (b) design patterns.*

change types. And finally, in Ovirt, “Renaming”, “Declaration”, and “Comment” are change types that leads DAPs to DPs mutation, while “Renaming”, “Operator”, and “Declaration” are the most representative change types in DPs to DAPs mutation.

Each of the mentioned change types may lead to a mutation. For example, “Import” makes the code confusing and less readable. Moreover, “Renaming” is the most frequent changed type. There are different classifications of renaming which has been described in (Arnaoudova *et al.*, 2014). In some cases, an entity like a package, type of classes, variables, constants, and parameters is renamed. In another case, one or more terms are changed (simple and complex renaming). Sometimes, when one or more terms of an identifier changes, the meaning of that also changes (semantic renaming). And finally, the grammar of an identifier is also changed during the renaming process (grammar renaming). When developers use a tool to apply a renaming operation, the tool may not rename all variables consistently in all the related files. However, certain source code changes should be made together to preserve consistency. For example, a parameter renaming affects all statements that reach the parameter in the method body. These statements must be adapted to the parameter change. Therefore, we suggest that developers should be careful when doing the renaming.

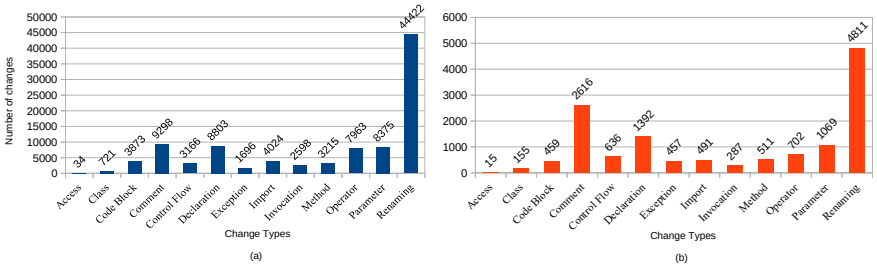

Figure 6.15 *Number of different types of changes in Apache Solr classes with (a) design anti-patterns and (b) design patterns.*

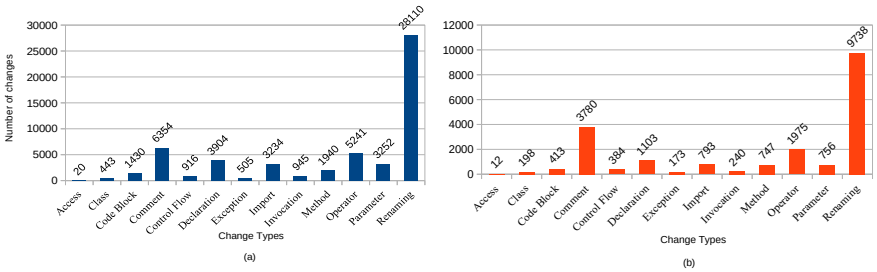

Figure 6.16 *Number of different types of changes in Mule classes with (a) design anti-patterns and (b) design patterns.*

In general, some of the change types affect the mutation from design patterns and–or design anti-patterns. We observe that the most representative change types leading to mutations in all the studied systems are “Renaming”, “Comment”, “Declaration”, and “Operator”.

6.3.3 RQ6.3: What is the fault-proneness of mutated design patterns and anti- patterns?What transitions lead to more fault-prone mutations?

Motivation Our results show that design patterns and design anti-patterns mutate during their evolution. However, to understand and assess the impacts of these mutations, we investigate these transitions to check whether the mutations are fault-prone. Moreover, it is important to know when, during evolution, design patterns and design anti-patterns lead to faults. Using this information, developers could understand the reasons of faults and take actions to reduce the risk of faults.

Analyzing design patterns and anti-patterns fault-proneness For each system, we mined the commit log and extracted bug numbers, commit ids related to those bugs, and the date when the bugs were introduced. We look for bugs introduced during evolution (from one snapshot to the next one), we use the dates to distinguish between bugs appearing in one snapshot and those appearing

between two extracted snapshots. We thus found the bugs introduced between each two consecutive snapshots for each system. Table 6.13 shows the number of faulty classes containing design patterns and anti-patterns for each system. We also computed the number of clean classes (bug-free classes) in the corresponding systems.

Our results (as shown in Table 6.13) show that classes that participate in design anti-patterns are more fault-prone than classes involved in design patterns. Eclipse is the only system that has more faulty classes than clean ones.

Table 6.13 Design anti-pattern and design-pattern mutations

Systems	**# of faulty classes**		**# of clean classes**
	Design Anti-patterns	Design Patterns	
Apache Ignite	10,984	1,051	81,093
Apache Solr	11,156	219	109,225
Eclipse	15,240	5,182	19,928
Matsim	4,053	1,888	896,510
Mule	17,794	5,924	197,574
Nuxeo	18,724	396	146,180
Ovirt	12,605	110	217,565

Analyzing transitions fault-proneness A mutation from design patterns and design anti-patterns can lead to faults. We match the buggy file names extracted from commit logs with the class names participating in both design patterns and design anti-patterns to identify the mutations experienced by these faulty classes. We are interested to know which design patterns and–or design anti-patterns experienced faulty mutations.

Table 6.14 presents the most representative mutations in each studied system, that led to faults. We observe that in most of the cases, transitions between design anti-patterns to design anti-patterns are more faulty. "LongParameterList" to 'LongMethod'" or "LongMethod" to "LazyClass" are some examples of such mutations in Apache Ignite. However, in Eclipse, Matsim, and Mule, there are mutations from design patterns to design patterns that led to more faults. "FactoryMethod" to "Decorator" in Eclipse, "Builder" to "FactoryMethod" in Matsim, and Mule are some of such transitions. In some cases, there is a transition between design anti-patterns and design patterns that led to faults as well, like "AntiSingleton" to "FactoryMethod" in Matsim. In Eclipse, we found a transition from "FactoryMethod" to "LongMethod"; which is a design pattern faulty mutation to design anti-patterns. Design anti-patterns are introduced by "bad" implementations or design choices and such choices are implementing or designing a (or part of a) class. This could make the classes very large and complex leading to comprehension overheads to the developers. On the other hand, design patterns are good solutions to solve the design and implementation problems in

the classes. Thus, the classes containing design anti-patterns are likely to be more fault-prone than classes containing design patterns; which is supported by our findings.

Table 6.14 Transitions Fault-proneness

System	Mutation Type	From	To	Probability
Apache Ignite	Design Anti-pattern → Design Anti-pattern	LongParameterList	LongMethod	57.1%
	Design Anti-pattern → Design Anti-pattern	LongMethod	LazyClass	28.5%
Apache Solr	Design Anti-pattern → Design Anti-pattern	RefusedParentBequest	MessageChain	42.7%
	Design Anti-pattern → Design Anti-pattern	LongMethod	LazyClass	15.6%
	Design Anti-pattern → Design Anti-pattern	ComplexClass	ClassDataShouldBePrivate	15.6%
Eclipe IDE	Design pattern → Design pattern	FactoryMethod	Decorator	49.2%
	Design Anti-pattern → Design Anti-pattern	LongMethod	LazyClass	38.5%
	Design pattern → Design Anti-pattern	FactoryMethod	LongMethod	5.6%
Matsim	Design pattern → Design pattern	Builder	FactoryMethod	67.7%
	Design Anti-pattern → Design Anti-pattern	SpagettiCode	RefusedParentBequest	15.2%
	Design Anti-pattern → Design pattern	AntiSingleton	FactoryMethod	11.4%
Mule	Design pattern → Design pattern	Builder	FactoryMethod	47.9%
	Design Anti-pattern → Design pattern	ComplexClass	FactoryMethod	26.4%
	Design Anti-pattern → Design Anti-pattern	ComplexClass	ClassDataShouldBePrivate	22.3%
Nuxeo	Design Anti-pattern → Design Anti-pattern	LazyClass	LargeClass	28.5%
	Design pattern → Design pattern	Singleton	FactoryMethod	49.5%
Ovirt	Design Anti-pattern → Design Anti-pattern	Blob	AntiSingleton	72.2%
	Design pattern → Design pattern	Singleton	Prototype	16.6%

Summary: *Design anti-patterns are more fault-prone than design patterns. Mutations from design anti-patterns to design patterns are more faulty than other types of mutations.*

6.3.4 RQ6.4: Do specific types of changes lead to increase fault-proneness during design patterns and-or design anti-patterns mutations?

Motivation Different types of changes have different impacts on the software systems due to the differences in extent of functional modification and the ripple effects of changes. Because of the varying complexity and impacts of changes, some changes are likely to introduce more faults compared to other types of changes. Thus, it is important to understand which types of changes increase the fault-proneness of the mutations of design patterns and design anti-patterns. In RQ6.4, we investigate whether specific types of changes lead to increased faults during design patterns and design anti-patterns evolution. This findings can help developers to be aware of specific change types to prevent the occurrence of faults during software evolution.

Analyzing change types leading to faults To answer this research question, we mined the committed files to extract the buggy file names for each system. As we explained in RQ6.3, for each system, we mined the commit log and extracted bug numbers, commit ids related to those bugs, and

the date when the bugs were introduced. We matched buggy files names with the same or similar class names in the changed files which also contain change types information, to find which classes that have been changed are also faulty. For an example of matching class names, if there is a class "a.b.c" and file "a.b.c.java", the class name and file name are matched. Besides, the class name "b.c" or "a.b.c" can also be matched with "a.b.c.java" and "b.c.java" file names respectively. For each system, we applied a script to identify the number of faulty classes that have changed. Table 6.15 presents the number of change types that led to faults. We found that in all studied systems, "Renaming", "Comment", and "Operator" are the most prevalent change types that lead to faults.

Table 6.15 Numbers of change types in the studied systems leading to faults

Systems →	**Apache Ignite**	**Apache Solr**	**Eclipse**	**Matsim**	**Mule**	**Nuxeo**	**oVirt**
Change Types ↓	# of changes	# of changes	# of changes	# of changes	# of changes	# of changes	# of changes
Access	18	18	37	22	11	62	25
Class	689	431	306	306	422	208	763
Code block	2,972	2,102	4,919	1,284	1,306	854	3,833
Comment	12,169	5,498	38,150	2,813	6,270	6,161	4,653
Control flow	2,903	1,935	11,678	660	1067	819	2,406
Declaration	5,912	5,386	11651	3,129	3,191	2,628	7,484
Exception	1,696	1221	1255	140	526	786	210
Import	2,831	2,400	2,958	1,425	2,443	2,064	4,268
Invocation	1,550	1,196	1,986	1,061	840	476	1,882
Method	2,509	1,851	4,093	637	1,697	1,229	3,619
Operator	5,120	4,364	16,094	6,215	4,228	2,675	8,134
Parameter	6,418	3,504	5,108	2,655	2,607	1,815	5,337
Renaming	47,811	24,640	67,040	32,445	22,968	12,736	65,245
Total changed classes	5,505	4,163	11,934	3,073	4,324	3,514	7,150

Analyzing the fault-proneness of classes with design patterns and design anti-patterns Table 6.16 presents the numbers of faulty changed classes and Figure 6.17 and Figure 6.18 present the percentages of faulty changed classes participating in design patterns and design anti-patterns, respectively for all the systems. We observe that change types have impacts on the fault-proneness of changed classes. Changed classes participating in design patterns are less faulty than those participating only in design anti-patterns. The results showed that some of the faulty classes are those which had changed in the past. For example, in Eclipse, the percentages of faulty classes participating in design patterns is 81% while for those participating in design anti-patterns it is 86%. The differences between these two categories are more visible in Apache Solr, where 51% of changed classes are participating in design anti-patterns, and only 11% of them have design patterns. In Rhino, changes impact fault-proneness significantly, because, on average, more than 85% of changed classes are faulty. Thus, we observe the trend that changed classes with design anti-patterns tend to be more fault-prone than changed classes with design patterns.

Table 6.16 Numbers of faulty and clean changed classes

Systems	Patterns	# Faulty classes	# Clean classes
Apache Ignite	Design Anti-patterns	5,112	4,178
	Design Patterns	393	464
Apache Solr	Design Anti-patterns	4,035	3,921
	Design patterns	128	1,064
Eclipse	Design Anti-patterns	9,406	1,551
	Design patterns	2,554	601
Matsim	Design Anti-patterns	2,549	30,042
	Design patterns	524	13,244
Mule	Design Anti-patterns	3,374	2,311
	Design patterns	950	1,225
Nuxeo	Design Anti-patterns	3,469	1,935
	Design patterns	45	36
oVirt	Design Anti-patterns	7,075	27,705
	Design patterns	75	482

Some change types make software systems more fault-prone compared to others. Our result show that in all the studied systems, “Renaming”, “Comment”, and “Operator” are the most prevalent change types that lead to faults.

6.3.5 RQ6.5: Do the occurrences of Linguistic anti-patterns increase change- and fault-proneness during design patterns and-or design anti-patterns mutations?

Motivation Linguistic Anti-patterns negatively impacts program comprehension which in turn may make the software maintenance tasks harder, leading to inconsistencies or faults. Moreover, it is important to know whether the existence of linguistic anti-patterns also add complexities to the evolution of design patterns and design anti-patterns; making the evolution more change- and fault-prone. Thus, in RQ6.5, we investigate whether the classes containing linguistic anti-patterns are more change- and fault-prone than mutated classes that contain only design patterns and design anti-patterns. This is important because developers can then focus and refactor these linguistic anti-patterns to prevent the software systems to be prone to changes and faults.

Analyzing impacts of linguistic anti-patterns on change-proneness As we discussed in RQ6.2, software systems undergo different types of changes during evolution. To analyze impacts of linguistic anti-patterns on change-proneness, we investigate whether the classes having linguistic anti-patterns are more change-prone compared to others. For each system, we compare the CSV files related to changed classes participating in design patterns and design anti-patterns with the detected linguistic design anti-patterns. The CSV files of the changed classes contain the name of

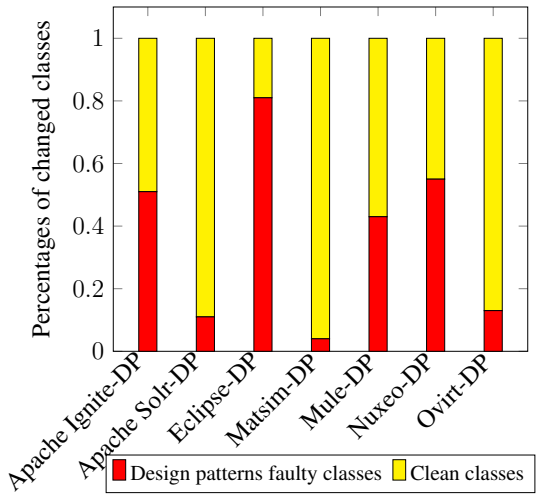

Figure 6.17 Faulty changed classes percentages with design pattern in the studied systems

the two subsequent snapshots where the change happened, the type of changes and the changed class names, and the number of changes. While the CSV file of linguistic anti-patterns contain the path of detected LA, the name of the class that has the LA, the type of detected LA, and finally the description of that LA. We map classes participating in occurrences of design patterns or design anti-patterns with classes that contain linguistic anti-patterns by comparing the names of classes. For each system, we applied a script to identify the number of changed classes that have linguistic anti-patterns. Table 6.17 shows the number of changed classes that contain linguistic anti-patterns along with design patterns or design anti-patterns.

Table 6.17 Change-prone classes with linguistic anti-patterns (LAs), design patterns(DPs) and design anti-patterns(DAPs)

Systems	**CCLA**	**DAPs and LAs**		**DPs and LAs**	
		CCAP	**CCLAAP**	**CCDP**	**CCLADP**
Apache Ignite	5,914	9,290	4,727	857	432
Apache Solr	6,369	7,956	2,830	1,192	729
Eclipse	446	10,957	129	3,155	38
Matsim	284	32,596	101	13,768	45
Mule	260	5,684	117	2,175	17
Nuxeo	430	5,402	290	81	10
Ovirt	517	34,780	248	557	7

CCLA, CCAP, CCDP= NO. of classes containing LAs, DAPs and DPs respectively
CCLAAP= NO. of changed classes containing LAs and DAPs,
CCLADP=NO. of changed classes containing LAs and DPs

The results show that the classes having both design anti-patterns and linguistic anti-patterns are more change-prone than the classes having just design anti-patterns. We observe the same for

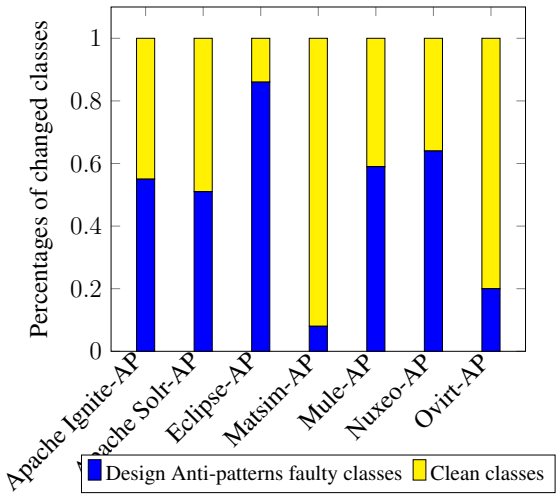

Figure 6.18 Faulty changed classes with design anti-patterns percentages in the studied systems

classes having both design patterns and linguistic anti-patterns. Based on Table 6.17 in Apache Ignite, 50% of changed classes that have design anti-patterns and–or design patterns also contain Linguistic anti-patterns.

Analyzing impacts of linguistic anti-patterns on fault-proneness For this analysis, as we mentioned in RQ6.3, for each system, we mined the committed files to extract the bug numbers and then matched with related commit ids to find the buggy file names, and the date when the bug was introduced during the evolution. Then, we match these file names with the same or even similar class names that have linguistic anti-patterns. We apply the same process as in Section 6.3.4. For each system, we applied a script to identify the number of faulty classes that have linguistic anti-patterns.

Table 6.18 Fault-proneness of classes with Linguistic Anti-patterns (LAs)

Systems	**NCLA**	**NFC**	**NFCLA**	**PFCLA**	**PCLAF**
Apache Ignite	5,914	3,303	438	13.26%	7.41%
Apache Solr	6,369	5,060	59	1.16%	0.91%
Eclipse	446	9,179	175	1.90%	39.24%
Matsim	284	747	14	1.87%	4.92%
Mule	260	5,176	78	1.50%	30%
Nuxeo	430	6,446	104	1.61%	24.19%
0Virt	517	698	11	1.58%	2.13%

NCLA = No. of classes containing LAs, **NFC** = No. of faulty classes
NFCLA = No. of faulty classes containing LAs
PFCLA = Percentage of faulty classes containing LAs
PCLAF = Percentage of classes containing LAs those are faulty

Mutated classes with linguistic anti-patterns and design anti-patterns are more change- and fault- prone than classes with only design anti-patterns. The same applies to the mutated classes that contain both linguistic anti-patterns and design patterns. We found that more than 50% of changed classes have both linguistic anti-patterns and design anti-patterns or design patterns.

6.4 Discussion

Different software systems may have different design patterns and–or design anti-patterns and they may evolve differently. From our analysis, we observe different mutation behavior for all analyzed systems because these systems have different designs, contexts, and development teams that modify the source code. Thus, they will have different design patterns and changes while following different mutation behaviors. We observed that some design patterns and–or design anti-patterns remained unchanged in all releases and they did not mutate during evolution.

For example, class *org.mule.test.infrastructure.process.MuleUtils* in all the snapshots of Mule, participates in “LongMethod” design anti-pattern. This design anti-pattern is introduced when developers continue adding new functionalities to a method while nothing is ever taken out. Usually, developers prefer to add code to an existing method instead of creating a new one (Brown *et al.*, 1998), which means that another line is added and then yet another, giving birth to a tangle of spaghetti code. This longer method or function will be very hard to understand and maintain.

We found that some of the design anti-patterns are mutated frequently to design patterns when developers are correcting faults during the evolution of the system. “Blob” is the most mutated design anti-pattern among the others in Apache Ignite system, which mutated to “AntiSingleton” with 37.5% probability. “Blob” presents a single class with a large number of attributes, operations, etc, surrounded by a number of data classes. The Blob Class is too complex for reuse and testing, while such classes are inefficient, and expensive to load into memory. There are also some design patterns mutated to design anti-patterns. “Command” is an example of design patterns that often mutates into “SwissArmyKnife" (i.e., 38.7% of the time) in the Matsim system. “SwissArmyKnife” is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, developers adds a large number of interface signatures in a useless attempt to meet all possible needs. In this case, the required code to create separate objects (“Command”) in a method was moved from the method to the code for invoking the method, but the created objects are passed to the method as parameters. Thus, the original class no longer knows about the relationships between objects, and dependency has decreased. But if several of these objects are created, each of them will require its parameter, which means a longer parameter list.

We observed that the major change types, which lead to more mutations, are “Renaming”, “Comment”, “Operator”, and “Declaration” in most of the studied systems. It means that these types of changes helps developers to correct the system and remove design anti-patterns as much as possible, so developers can just focus on making such changes to improve the quality. It is interesting to discover what types of changes make the system more fault-prone. We observed that design anti-patterns are more fault-prone than design patterns. We noticed that in the case of mutation, the most representative case was “LongParameterList” to “LongMethod”. In this case, a design pattern is frequently used in the source code where it is not beneficial. Introducing unnecessary restrictions is inefficient in situations where an individual instance of a class is not required.

Using the obtained results developers could know which types of LAs, DAPs, and DPs are more change - and fault-prone, so they can focus on refactoring those smells and patterns. Besides, understanding which types of mutations lead to more changes and faults could help developers prioritize their refactoring and code improvement operations. They could avoid mutations that are likely to induced more changes and faults.

6.5 Threats to Validity

We now discuss the potential threats to the validity of the findings of our study following existing guidelines by (Yin, 2013; Wohlin *et al*., 2012).

Construct validity threats concern the relation between theory and observation. We know that the used design pattern and design anti-pattern detection techniques (DECOR) and (DeMIMA) in this study include some subjective understanding related to the definition of design patterns and design anti-patterns. The authors reported that the recall rate is 100% for both techniques while the precision rate in the worst case is 31% and the average is 60% for DECOR and 34% and 80% respectively for DeMIMA. Besides, we accept that the precision of these techniques is a concern. Some false positive classes may pass the validation because they “looks like” a motif. Besides, we accept that those methods just consider the most common techniques to implement DPs, and rules to define DAPs, although developers do not implement DAPs, DPs, and even LAs exactly as they are defined in the references. They usually implement them based on their needs and even use variants of them. We accept that in finding change types which led to a fault, we could have matched class names that are not representing the same class. For example, class “c” is not match with “a.b.c.java” but it could be matched with the “b.c.java” file. Moreover, we know that during the evolution, the name of the classes may change as well. As for precision, the manual validation could be affected by subjectiveness or human error. We should consider each type of renaming. Because in this case, we may misinterpret that there is a mutation from design patterns to design anti-patterns and vice versa, while in fact the name of the class has been changed and these patterns

remained stable. We also used the LADP-plugin to detect linguistic anti-patterns because it has been implemented based on the most novel and recent approach by (Arnaoudova *et al.*, 2013). Other possible design smells detection techniques (e.g., inFusion, JDeodorant or PMD) can be used to confirm our findings.

Internal validity threats concerns factors affecting the results. This threat is about the causality drawn from the study. It concerns our selection of analyzed systems and analysis method. The accuracy of DECOR and DeMIMA impacts our results, since the number of design patterns and design anti-patterns computed with DECOR and DeMIMA is used to calculate the probability of mutations. Other detection techniques and tools should be applied to validate our findings.

Conclusion validity threats concern the relationship between the treatment and the results. We paid attention in choosing the system sizes. We used the SZZ algorithm (Śliwerski *et al.*, 2005) to identify the commits that introduced a fault. Although this algorithm may yield false positive results, it has been successfully employed in previous works, such as (Kamei *et al.*, 2013; Fukushima *et al.*, 2014), with satisfying results. In this thesis, to increase the algorithm's accuracy, we removed all fault-inducing commit candidates that only changed blank or comment lines. Moreover, the static analysis tool, srcML, can identify about 100 types of code elements from source code. To make our results more actionable for software practitioners, we manually grouped similar element tags into 12 major change types as shown in Table 6.2, which can help software practitioners carefully change and review fault-prone code. Besides, as we mentioned before, there are false positives among the design patterns and design anti-patterns detected using DECOR and DeMIMA.

Reliability Validity threats concern the possibility of replicating this study. We studied snapshots of every 500 commits in seven open-source systems but do not claim that these results are representative of all systems or developers. We provide all the necessary data on-line[16] to help other researchers replicate our work.

External validity threats concern the ability to generalize our results. We studied seven software systems with different sizes, domains, and complexity. However, all of them are written in Java, they are open source, and they are available on-line. In the future, we plan to investigate more diverse set of subject systems. Moreover, we also aim to focus on larger projects, with other programming languages, such as C++. We choose thirteen design anti-patterns and six design patterns among the many available patterns. The results could be different with industrial systems, other object-oriented programming languages, and different patterns that we plan to investigate in the future.

[16] `http://www.ptidej.net/downloads/replications/EMSE19a/`

6.6 Summary

The chapter presented an empirical study investigating the behaviors and impacts of linguistic anti-patterns, design patterns and design anti-patterns in terms of change- and fault-proneness during software evolution.

The results show that design patterns and design anti-patterns mutate into one another during software evolution and that these mutations impact the fault-proneness of classes participating in the design (anti-)patterns. These mutations impacted the quality of studied systems in terms of change- and fault-proneness. When a mutation led to the introduction of another design anti-pattern, the quality decreased in terms of fault-proneness. When design anti-patterns were removed or mutated into design patterns, the quality increased in terms of change- and fault-proneness.

Besides, classes containing both linguistic anti-patterns and design anti-patterns are more change-prone than classes that just have design anti-patterns. There is the same observations for classes containing both linguistic anti-patterns and design patterns. In terms of fault-proneness, we found that the occurrences of linguistic anti-patterns to some extent, can increase the probability of faults in systems.

CHAPTER 7 CONCLUSION

7.1 Dissertation Findings and Conclusions

This dissertation reports about three empirical study that investigate the impact of linguistic anti-patterns on the code quality. We also study the impact of both linguistic anti-patterns and design anti-patterns–or design patterns on change- and fault-proneness. We made the following observations:

First, regarding the impact of the occurrences of LAs on the code understandability, we observe that LAs negatively affect participants' understanding; decreasing the numbers of correct answers. We also study the role that prior knowledge of LAs has on the effect of LA occurrences on code understandability and observe that prior knowledge can mitigate the negative impact of LAs.

Second, we performed another empirical study on design anti-patterns and linguistic anti-patterns impact on quality, measured in terms of change- and fault- proneness. Our investigation consisted in the analysis of 30 releases of three different open-source systems: ArgoUML, Hibernate, and ANT. We detected 29 smells in each release, i.e., 12 design anti-patterns using the DECOR approach and 17 LAs using the LDAP approach. To study the relation between the detected families of smells and change- and fault-proneness, we used the change history of the studied systems extracted from their Git/SVN version control systems. We also mined their bug repositories. The results show that Linguistic anti-patterns alone do not contribute much to the change- proneness of classes of object-oriented systems. In addition, they indicate that, in a lot of cases, classes containing design anti-patterns are more change- and fault-prone than classes with linguistic anti-patterns only. The occurrence of design anti-patterns in a class that experienced a linguistic anti-patterns has a strong relationship with change- and fault-proneness than the occurrence of linguistic anti-patterns in a class that experienced a design anti-pattern.

In the last study, we examined design anti-pattern and design pattern evolution and the impact of this evolution on code quality. The main goal is to help software developers in non-trivial tasks related to code evolution analysis by modeling design anti-pattern mutations. Our study was performed on seven different open-source systems: Apache Ignite, Apache Solr, Eclipse, Matsim, Mule, Nuxeo, and Ovirt. We detected 13 design anti-patterns using DECOR and 8 design patterns using DeMIMA in each system. The results showed that design anti-patterns mutate to represent other forms of more complicated design anti-patterns or even design patterns. Moreover, our findings show that design anti-pattern classes that mutated over the evolution of systems are significantly less fault-prone than non-mutated classes while design patterns are less fault-prone than

design anti-patterns in general. We also investigate the change types that occurred during the evolution of the studied systems. Finally, we found that classes containing both linguistic anti-patterns and design anti-patterns are more change-prone than classes that just have design anti-patterns. We have the same observation for classes containing both linguistic anti-patterns and design patterns. In terms of fault-proneness, we found that the occurrences of linguistic anti-patterns to some extent, can increase the probability of faults in the studied systems.

Using this information, developers can focus on the design patterns that are most likely to mutate into design anti-patterns and–or to experience more faults. Thus, this information can help development and quality assurance teams to better focus their refactoring efforts on classes with design patterns and design anti-patterns that could mutate into patterns with higher change- and–or fault-proneness. This is likely to be useful to control and improve the quality of their systems.

7.2 Future Directions

Our work opens several new research directions. We outline some of them as follows:

LAs and Code Quality

In the future, we plan to study other types of LAs to see their impact on the code understandability. We also want to reproduce our controlled experiment with other participants, in particular from the industry where participants are asked to understand part of source code of a project with LA and a version where the LA was improved or refactored. To evaluate the impact, an eye-tracking system could be used to measure the degree of understanding, the time that participants take to answer correctly, and the effort that they do to understand. We also want to improve LAs detection tools using insights gathered during this study. Finally, we want to study other programming languages, specifically dynamic programming languages, in which developers rely even more on names to compensate for the lack of types.

LAs, DAPs and change- and fault-proneness

As future work, we intend to conduct a user study involving professional developers both internal, i.e., contributors to the development of the systems as well as external ones from industry to better understand the interaction between design anti-patterns and linguistic anti-patterns, and identify which specific type of DAPs (e.g., SpaghettiCode) and LAs (e.g., Attribute signature and comment are opposite) should be given higher priority during refactoring.

LAs, DAPs, DPs mutations and change- and fault-proneness

In the near future, we plan to study the reason for the emergence of faults after design anti-pattern and design-pattern mutation. We want to mine expertise and time pressure information from version

control systems and release documents, which contains more information about the developers and their activities that could be the cause of faults after design anti-pattern and design pattern mutate.

Moreover, building Markov models for LAs mutation to DAPs could be considered as another future work. Studying the software evolution to see how their mutations impacted the change- and fault-proneness of the classes participating in these design anti-patterns.

Besides, it is interesting to investigate Which type(s) of linguistic anti-patterns, design anti-patterns and design-patterns are more risky and may leads to become high severity fault. To do so, we will categorize faults across different severity levels and examine which types of LAs, DAPs, and DPs mutations are related to faults with high severity. This will be useful to help development teams prioritize their limited resources toward more impactful design smells.

Bibliography

Abbes, Marwen and Khomh, Foutse and Gueheneuc, Yann-Gael and Antoniol, Giuliano (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE, 181–190.

Abebe, Surafel Lemma and Arnaoudova, Venera and Tonella, Paolo and Antoniol, Giuliano and Gueheneuc, Yann-Gael (2012). Can lexicon bad smells improve fault prediction? *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 235–244.

Abebe, Surafel Lemma and Haiduc, Sonia and Tonella, Paolo and Marcus, Andrian (2009). Lexicon bad smells in software. *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 95–99.

Abebe, Surafel Lemma and Haiduc, Sonia and Tonella, Paolo and Marcus, Andrian (2011). The effect of lexicon bad smells on concept location in source code. *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. Ieee, 125–134.

Abebe, Surafel Lemma and Tonella, Paolo (2013). Automated identifier completion and replacement. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 263–272.

An, Le and Khomh, Foutse (2015). An empirical study of crash-inducing commits in mozilla firefox. *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 5.

Anquetil, Nicolas and Lethbridge, Timothy (1998). Assessing the relevance of identifier names in a legacy software system. *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 4.

Arnaoudova, Venera and Di Penta, Massimiliano and Antoniol, Giuliano (2016). Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, *21*(1), 104–158.

Arnaoudova, Venera and Di Penta, Massimiliano and Antoniol, Giuliano and Gueheneuc, Yann-Gael (2013). A new family of software anti-patterns: Linguistic anti-patterns. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 187–196.

Arnaoudova, Venera and Eshkevari, Laleh M and Di Penta, Massimiliano and Oliveto, Rocco and Antoniol, Giuliano and Gueheneuc, Yann-Gael (2014). Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, *40*(5), 502–532.

Bieman, James M and Straw, Greg and Wang, Huxia and Munger, P Willard and Alexander, Roger T (2003). Design patterns and change proneness: An examination of five evolving systems. *Software metrics symposium, 2003. Proceedings. Ninth international*. IEEE, 40–49.

Brown, William H and Malveau, Raphael C and McCormick, Hays W and Mowbray, Thomas J (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Canfora, Gerardo and Cerulo, Luigi and Di Penta, Massimiliano and Pacilio, Francesco (2010). An exploratory study of factors influencing change entropy. *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 134–143.

Caprile, Bruno and Tonella, Paolo (2000). Restructuring program identifier names. *icsm*. 97–107.

Caprile, C and Tonella, Paolo (1999). Nomen est omen: Analyzing the language of function identifiers. *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. IEEE, 112–122.

De Lucia, Andrea and Di Penta, Massimiliano and Oliveto, Rocco (2011). Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, *37*(2), 205–227.

Dreyfus, Stuart E and Dreyfus, Hubert L (1980). A five-stage model of the mental activities involved in directed skill acquisition. Rapport technique, University of California Berkley.

Fakhoury, Sarah and Ma, Yuzhan and Arnaoudova, Venera and Adesope, Olusola (2018). The effect of poor source code lexicon and readability on developers' cognitive load. *Proceedings of the 26th Conference on Program Comprehension*. ACM, New York, NY, USA, ICPC '18, 286–296.

Fischer, Michael and Pinzger, Martin and Gall, Harald (2003). Populating a release history database from version control and bug tracking systems. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 23–32.

Fukushima, Takafumi and Kamei, Yasutaka and McIntosh, Shane and Yamashita, Kazuhiro and Ubayashi, Naoyasu (2014). An empirical study of just-in-time defect prediction using cross-project models. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 172–181.

Gamma, Erich (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Gatrell, Matt and Counsell, Steve and Hall, Tracy (2009). Design patterns and change proneness: a replication using proprietary c# software. *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 160–164.

Guéhéneuc, Yann-Gaël and Antoniol, Giuliano (2008). Demima: A multilayered approach for design pattern identification. *Software Engineering, IEEE Transactions on*, *34*(5), 667–684.

Gueheneuc, Yann-Gael and Sahraoui, Houari and Zaidi, Farouk (2004). Fingerprinting design patterns. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 172–181.

Guerrouj, Latifa and Kermansaravi, Zeinab and Arnaoudova, Venera and Fung, Benjamin CM and Khomh, Foutse and Antoniol, Giuliano and Guéhéneuc, Yann-Gaël (2015). Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. *Software Quality Journal*, 1–30.

Hart, Sandra G and Staveland, Lowell E (1988). Development of nasa-tlx (task load index): Results of empirical and theoretical research. *Advances in psychology*, Elsevier, vol. 52. 139–183.

Hassan, Ahmed E (2009). Predicting faults using the complexity of code changes. *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 78–88.

Hofmeister, Johannes and Siegmund, Janet and Holt, Daniel V (2017). Shorter identifier names take longer to comprehend. *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 217–227.

Iacob, Claudia (2011). A design pattern mining method for interaction design. *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 217–222.

Jaafar, Fehmi and Guéhéneuc, Yann-Gaël and Hamel, Sylvie (2013). Analysing anti-patterns static relationships with design patterns. *Proc. PPAP*, *2*, 26.

Jaafar, Fehmi and Khomh, Foutse and Guéhéneuc, Yann-Gaël and Zulkernine, Mohammad (2014). Anti-pattern mutations and fault-proneness. *Quality Software (QSIC), 2014 14th International Conference on*. IEEE, 246–255.

Kamei, Yasutaka and Shihab, Emad and Adams, Bram and Hassan, Ahmed E and Mockus, Audris and Sinha, Aloka and Ubayashi, Naoyasu (2013). A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, *39*(6), 757–773.

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

Khomh, Foutse and Guéhéneuc, Yann-Gaël (2008). Do design patterns impact software quality positively? *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 274–278.

Khomh, Foutse and Guéhéneuc, Yann-Gaël and Antoniol, Giuliano (2009a). Playing roles in design patterns: An empirical descriptive and analytic study. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 83–92.

Khomh, Foutse and Vaucher, Stéphane and Guéhéneuc, Yann-Gaël and Sahraoui, Houari (2009b). A bayesian approach for the detection of code and design smells. *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 305–314.

Krämer, Christian and Prechelt, Lutz (1996). Design recovery by automated search for structural design patterns in object-oriented software. *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. IEEE, 208–215.

Lanza, Michele and Marinescu, Radu (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

Marinescu, Radu and Lanza, Michelle (2006). Object-oriented metrics in practice.

McIntosh, Shane and Kamei, Yasutaka and Adams, Bram and Hassan, Ahmed E (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 192–201.

McIntosh, Shane and Kamei, Yasutaka and Adams, Bram and Hassan, Ahmed E (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189.

Merlo, Ettore and McAdam, Ian and De Mori, Renato (2003). Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, *15*(4), 205–244.

Meyn, Sean P and Tweedie, Richard L (2012). *Markov chains and stochastic stability*. Springer Science & Business Media.

Moha, Naouel and Gueheneuc, Yann-Gael and Duchien, Laurence and Le Meur, Anne-Francoise (2010). Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, *36*(1), 20–36.

Olbrich, Steffen and Cruzes, Daniela S and Basili, Victor and Zazworka, Nico (2009). The evolution and impact of code smells: A case study of two open source systems. *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 390–400.

D. Rapu and S. Ducasse and T. Girba and R. Marinescu (2004). Using history information to improve design flaws detection. *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* 223–232.

Rapu, D and Ducasse, Stéphane and Gîrba, Tudor and Marinescu, Radu (2004). Using history information to improve design flaws detection. *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. IEEE, 223–232.

Riel, Arthur J (1996). *Object-oriented design heuristics*, vol. 335. Addison-Wesley Reading.

Romano, Daniele and Raila, Paulius and Pinzger, Martin and Khomh, Foutse (2012). Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 437–446.

Settas, Dimitrios and Cerone, Antonio and Fenz, Stefan (2012). Enhancing ontology-based antipattern detection using bayesian networks. *Expert Systems with Applications*, *39*(10), 9041–9053.

Sheskin, David J (2003). *Handbook of parametric and nonparametric statistical procedures*. crc Press.

Shull, Forrest and Singer, Janice and Sjøberg, Dag IK (2007). *Guide to advanced empirical software engineering*. Springer.

Sillito, Jonathan and Murphy, Gail C and De Volder, Kris (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451.

Śliwerski, Jacek and Zimmermann, Thomas and Zeller, Andreas (2005). When do changes induce fixes? *ACM sigsoft software engineering notes*, *30*(4), 1–5.

Soloway, Elliot and Bonar, Jeffrey and Ehrlich, Kate (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, *26*(11), 853–860.

srcml (2016). srcML. `http://www.srcml.org`. Online; Accessed March 31st, 2016.

Taba, Seyyed Ehsan Salamati and Khomh, Foutse and Zou, Ying and Hassan, Ahmed E and Nagappan, Meiyappan (2013). Predicting bugs using antipatterns. *2013 IEEE International Conference on Software Maintenance*. IEEE, 270–279.

Tan, Lin and Yuan, Ding and Krishna, Gopal and Zhou, Yuanyuan (2007). icomment: Bugs or bad comments? *ACM SIGOPS Operating Systems Review*, *41*(6), 145–158.

Tan, Lin and Zhou, Yuanyuan and Padioleau, Yoann (2011). acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 11–20.

Tan, Shin Hwei and Marinov, Darko and Tan, Lin and Leavens, Gary T (2012). @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 260–269.

Toutanova, Kristina and Manning, Christopher D (2000). Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*. Association for Computational Linguistics, 63–70.

Tsantalis, Nikolaos and Chatzigeorgiou, Alexander and Stephanides, George and Halkidis, Spyros T (2006). Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, *32*(11), 896–909.

Van Emden, Eva and Moonen, Leon (2002). Java quality assurance by detecting code smells. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 97–106.

Vaucher, Stephane and Khomh, Foutse and Moha, Naouel and Guéhéneuc, Yann-Gaël (2009). Tracking design smells: Lessons from a study of god classes. *16th Working Conference on Reverse Engineering (WCRE 2009), IEEE Computer Society Press (WCRE'09)*. 145–154.

Vlissides, John and Helm, Richard and Johnson, Ralph and Gamma, Erich (1995). Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, *49*(120), 11.

Vokáč, Marek (2004). Defect frequency and design patterns: An empirical study of industrial code. *Software Engineering, IEEE Transactions on*, *30*(12), 904–917.

Webster, Bruce F (1995). *Pitfalls of object oriented development*. M\ & T Books.

Wohlin, Claes and Runeson, Per and Höst, Martin and Ohlsson, Magnus C and Regnell, Björn and Wesslén, Anders (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yamashita, Aiko and Moonen, Leon (2013). Do developers care about code smells? an exploratory survey. *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 242–251.

Yin, Robert K (2013). *Case study research: Design and methods*. Sage publications.