UNIVERSITÉ DE MONTRÉAL

TOWARDS IMPROVING THE CODE LEXICON AND ITS CONSISTENCY

VENERA ARNAOUDOVA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AOÛT 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

TOWARDS IMPROVING THE CODE LEXICON AND ITS CONSISTENCY

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

M. MERLO Ettore, Ph.D., président
M. ANTONIOL Giuliano, Ph.D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doct., membre et codirecteur de recherche
M. ADAMS Bram, Doct., membre
Mme POLLOCK Lori L., Ph.D., membre

To Julien...

ACKNOWLEDGMENTS

I am grateful to my supervisor, Giulio, for everything I learned from him, for all the opportunities he gave me, for his guidance when I was lost, for his patience when I needed him to listen, for being honest when he had doubts about ideas and methodologies, for being not so honest when answering *"I don't know"* because he wanted me to figure it out, but mostly, thank you for challenging me ! He thought me the real meaning of SW and WC [1].

I am grateful to my co-supervisor, Yann, for his comments and feedback.

I also express my special gratitude to Max (Prof. Massimiliano Di Penta) for giving me the opportunity to work with him, for helping me to develop my ideas and leading them to end. Even in panic mode he always had by back, found time to provide feedback, work, and discuss new ideas.

During my Ph.D. I also had the chance to work with great researchers from whom I learned a lot. I thank Rocco (Prof. Rocco Oliveto) for sharing great ideas. I thank Paolo (Prof. Paolo Tonella) for giving me the opportunity to collaborate and visit his team in Trento. I thank Surafel (Dr. Surafel Lemma Abebe) for the several weeks we worked together.

I also thank all friends and colleagues, current and past members of the Soccerlab, Ptidej, Swat, and Mcis teams, for sharing ideas, providing feedback, and of course playing badminton. Thank you Laleh (almost Dr. Laleh Eshkevari) for being like a sister to me. Thank you for being by my side all these years, for your support, for all the passionate discussions we had, and for the sleepless nights we spend working.

I am very thankful to my 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. I love you all so much !

Last but certainly not least, I thank my beloved husband Julien 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 !

1. SW and WC stand for Giulio's most frequently asked questions : *"So What ?"* and *"Who Cares ?"*

RÉSUMÉ

La compréhension des programmes est une activité clé au cours du développement et de la maintenance des logiciels. Bien que ce soit une activité fréquente—même plus fréquente que l'écriture de code—la compréhension des programmes est une activité difficile et la difficulté augmente avec la taille et la complexité des programmes. Le plus souvent, les mesures structurelles—telles que la taille et la complexité—sont utilisées pour identifier ces programmes complexes et sujets aux bogues. Cependant, nous savons que l'information linguistique contenue dans les identifiants et les commentaires—c'est-à-dire le lexique du code source—font partie des facteurs qui influent la complexité psychologique des programmes, c'est-à-dire les facteurs qui rendent les programmes difficiles à comprendre et à maintenir par des humains.

Dans cette thèse, nous apportons la preuve que les mesures évaluant la qualité du lexique du code source sont un atout pour l'explication et la prédiction des bogues. En outre, la qualité des identifiants et des commentaires peut ne pas être suffisante pour révéler les bogues si on les considère en isolation—dans sa théorie sur la compréhension de programmes par exemple, Brooks avertit qu'il peut arriver que les commentaires et le code soient en contradiction. C'est pourquoi nous adressons le problème de la contradiction et, plus généralement, d'incompatibilité du lexique en définissant un catalogue d'Antipatrons Linguistiques (LAs), que nous définissons comme des mauvaises pratiques dans le choix des identifiants résultant en incohérences entre le nom, l'implémentation et la documentation d'une entité de programmation. Nous évaluons empiriquement les LAs par des développeurs de code propriétaire et libre et montrons que la majorité des développeurs les perçoivent comme mauvaises pratiques et par conséquent elles doivent être évitées. Nous distillons aussi un sous-ensemble de *LAs canoniques* que les développeurs perçoivent particulièrement inacceptables ou pour lesquelles ils ont entrepris des actions. En effet, nous avons découvert que 10% des exemples contenant les LAs ont été supprimés par les développeurs après que nous les leur ayons présentés.

Les explications des développeurs et la forte proportion de LAs qui n'ont pas encore été résolus suggèrent qu'il peut y avoir d'autres facteurs qui influent sur la décision d'éliminer les LAs, qui est d'ailleurs souvent fait par le moyen de renommage. Ainsi, nous menons une enquête auprès des développeurs et montrons que plusieurs facteurs peuvent empêcher les développeurs de renommer. Ces résultats suggèrent qu'il serait plus avantageux de souligner les LAs et autres mauvaises pratiques lexicales quand les développeurs écrivent du code source—par exemple en utilisant notre plugin LAPD Checkstyle détectant des LAs—de sorte que l'amélioration puisse se faire sur la volée et sans impacter le reste du code.

ABSTRACT

Program comprehension is a key activity during software development and maintenance. Although frequently performed—even more often than actually writing code—program comprehension is a challenging activity. The difficulty to understand a program increases with its size and complexity and as a result the comprehension of complex programs, in the best-case scenario, more time consuming when compared to simple ones but it can also lead to introducing faults in the program. Hence, structural properties such as size and complexity are often used to identify complex and fault prone programs. However, from early theories studying developers' behavior while understanding a program, we know that the textual information contained in identifiers and comments—i.e., the source code lexicon—is part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans.

In this dissertation we provide evidence that metrics evaluating the quality of source code lexicon are an asset for software fault explanation and prediction. Moreover, the quality of identifiers and comments considered in isolation may not be sufficient to reveal flaws—in his theory about the program understanding process for example, Brooks warns that it may happen that comments and code are contradictory. Consequently, we address the problem of contradictory, and more generally of inconsistent, lexicon by defining a catalog of Linguistic Antipatterns (LAs), i.e., poor practices in the choice of identifiers resulting in inconsistencies among the name, implementation, and documentation of a programming entity. Then, we empirically evaluate the relevance of LAs—i.e., how important they are—to industrial and open-source developers. Overall, results indicate that the majority of the developers perceives LAs as poor practices and therefore must be avoided. We also distill a subset of *canonical LAs* that developers found particularly unacceptable or for which they undertook an action. In fact, we discovered that 10% of the examples containing LAs were removed by developers after we pointed them out.

Developers' explanations and the large proportion of yet unresolved LAs suggest that there may be other factors that impact the decision of removing LAs, which is often done through renaming. We conduct a survey with developers and show that renaming is not a straightforward activity and that there are several factors preventing developers from renaming. These results suggest that it would be more beneficial to highlight LAs and other lexicon bad smells as developers write source code—e.g., using our LAPD Checkstyle plugin detecting LAs—so that the improvement can be done on-the-fly without impacting other program entities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CK	Chidamber and Kemerer object-oriented metrics suite
FPA	Fault Percentile Average
HEHCC	High Entropy and High Context Coverage
IDE	Integrated Development Environment
IR	Information Retrieval
LA	Linguistic Antipattern
LAPD	Linguistic AntiPattern Detector
LBS	Lexicon Bad Smells
LSI	Latent Semantic Indexing
LOC	Lines Of Code
LRM	Logistic Regression Model
MCC	Matthew's Correlation Coefficient
NLPA	Natural Language Program Analysis
OO	Object Oriented
OR	Odd Ratio
PCA	Principal Component Analysis
POM	Primitives Operators Metrics
POS	Part Of Speech
RF	Random Forest
SLOC	Source Lines Of Code
SVM	Support Vector Machine
VCS	Version Control System
VIF	Variance Inflation Factors

CHAPTER 1

INTRODUCTION

1.1 Context

Previous studies show that the main cost for the development of a software is spent on its maintenance (Lientz *et al.*, 1978; Lehman, 1980). Studies also show that during maintenance, developers spend the majority of their time understanding source code (von Mayrhauser *et al.*, 1997; Standish, 1984; Tiarks, 2011). This leads to the observation that program comprehension is a key activity during the software life cycle. To understand the source code, developers follow different exploration strategies. Researchers have been observing how developers explore code to study the different strategies and their effectiveness. Brooks (1983) presents the process of program understanding as a top-down hypothesis driven approach in which an initial and vague hypothesis is formulated—based on the developer's knowledge about the program domain or other related domains—and incrementally refined into more specific hypotheses based on the information extracted from the program text and documentation. Soloway *et al.* (1988) observe that a systematic strategy, i.e., reading code line-by-line, always results in correct enhancement whereas an opportunistic strategy results in correct enhancement only half of the time. Corritore et Wiedenbeck (2001) observe that Object Oriented (OO) programmers follow a top-down approach during early familiarisation with the program but increasingly use a bottom-up approach in subsequent after. Robillard *et al.* (2004) observe that a methodical investigation is more effective than the opportunistic one.

Regardless of which strategy developers use, they spend a considerable amount of time reading program identifiers while exploring the source code. In fact, previous studies show that more than 70% of the source code of a software consists of program identifiers (Deißenböck et Pizka, 2005) and, more important, that the source code contains 42% of the domain terms (Haiduc et Marcus, 2008). This confirms the hypothesis of Brooks (1983) that identifiers and comments are part of the internal indicators for the meaning of a program. The knowledge contained in source code identifiers is even more valuable when no other source of documentation exists. Consequently, there has been a large amount of works exploring the information contained in program identifiers and comments for various purposes. Pollock *et al.* (2007) refer to this as Natural Language Program Analysis (NLPA) and define it as the combination of *natural language processing techniques with program analysis to extract natural language information from the identifiers, literals, and comments of a program.*

For example, several approaches reconstruct a program domain knowledge by extracting concepts from source code identifiers and–or comments (Anquetil et Lethbridge, 1998; Merlo *et al.*, 2003; Falleri *et al.*, 2010; Abebe et Tonella, 2011). Others use identifiers and program analysis to automatically generate source code documentation at different granularity level—e.g., block (Sridhara *et al.*, 2011), method (Sridhara *et al.*, 2010), and class (Moreno *et al.*, 2013)—or to improve the code readability by inserting blank lines between different algorithmic steps (Wang *et al.*, 2014). Some researchers show that the semantics carried by identifiers can be used to perform refactoring and re-modularization (Bavota *et al.*, 2011, 2013).

Thus, it is reasonable to believe that the quality of the source code lexicon is of paramount importance for program comprehension and any NLPA technique. On the basis of several experiments, Shneiderman et Mayer (1975) observe a significantly better program comprehension by subjects using commented programs. A higher number of subjects located bugs in commented programs than in not commented programs, although the difference is not statistically significant. They argue that program comments and mnemonic identifiers simplifies the conversion process from the program syntax to the program internal semantic representation. Chaudhary et Sahasrabuddhe (1980) argue that the psychological complexity of a program—i.e., the characteristics that make a program difficult to understand by humans—is an important aspect of program quality. They identify several features that contribute to the psychological complexity one of which is termed "meaningfulness". They argue that meaningful variable names and comments facilitate program understanding as they facilitate the relation between the program semantics and the problem domain. An experiment with students using different versions of FORTRAN programs—with and without meaningful names—confirmed the hypothesis.

Later, researchers deepened the studies on the nature of identifiers, i.e., their internal structure (Caprile et Tonella, 1999, 2000), and on the use of abbreviations and single-letters and their impact for program comprehension (Lawrie *et al.*, 2006b, 2007b). Specific tools have been developed to leverage the knowledge carried by identifiers and comments. For example, researchers propose different techniques for identifier splitting (Enslen *et al.*, 2009; Lawrie *et al.*, 2010; Guerrouj *et al.*, 2013), expansion (Lawrie et Binkley, 2011; Guerrouj *et al.*, 2013), and Part Of Speech (POS) tagging (Binkley *et al.*, 2011; Abebe et Tonella, 2010; Gupta *et al.*, 2013).

Deißenböck et Pizka (2005) also define guidelines to construct high-quality identifiers and Lawrie *et al.* (2006a) propose ways to detect violations of those guidelines. Inspired form code smells (Fowler, 1999), Abebe *et al.* (2009b) define a catalog of Lexicon Bad Smells (LBS) defined as *potential lexicon construction problems, which could be solved by means of*

refactoring (typically renaming) actions. Abebe *et al.* (2011) show that such LBS negatively impacts Information Retrieval (IR)-based concept location. Previous research also shows that the quality of identifiers is an important factor for the quality of the project (Butler *et al.*, 2009, 2010; Buse et Weimer, 2010; Poshyvanyk et Marcus, 2006; Marcus *et al.*, 2008).

However, sometimes the quality of identifiers considered in isolation may not be sufficient to reveal flaws and one may need to consider the consistency among identifiers, documentation, and implementation. In his theory about the process that programmers follow to understand a program, Brooks (1983) explains that while trying to refine or verify a hypothesis developers will sometimes need to inspect the code in detail, e.g., check the comments against the code. Brooks warns that it may happen that comments and code are contradictory and that the decision of which indicator to trust (i.e., comment or code) primarily depends on the overall support of the hypothesis being tested rather than the type of the indicator itself. This implies that when contradiction between code and comments occur, different developers may trust different indicators and thus have different interpretations of a program, which may result in faults. Zhong *et al.* (2011) reveal real faults by automatically mining resource usage specifications from Application Programming Interface (API) documentation and checking them against the source code. Other approaches identify inconsistencies related to resource locks and function calls (Tan *et al.*, 2007), synchronizations (Tan *et al.*, 2011), null values and exceptions (Tan *et al.*, 2012), actual and formal method parameters (Pradel et Gross, 2013), lexicon of source code and high-level artifacts (De Lucia *et al.*, 2011), and method name and data/control-flow properties (Høst et Østvold, 2009).

The ultimate goal of defining measures for poor lexicon is to increase developers' awareness of the existence of such practices and help them to improve the quality of the source code lexicon. In other words, we, researchers, hypothesize that developers will undertake an action and, for example, rename a method if its name does not any more reflect its functionality. However, from a study that we performed on LAs with developers of several project, we observe that they removed LAs in 10% of the cases we pointed them out suggesting that there may be other factors that prevent developers to rename. Antoniol *et al.* (2007) also observed that the evolution of the source code lexicon is more stable compared to the structural evolution of a project, i.e., the lexicon evolves less than the structure.

1.2 Problem Statement and Contributions

The research summarized above steered the definition of our thesis.

> Our thesis is that poor lexicon negatively impacts the quality of software, that the quality of the lexicon depends on the quality of individual identifiers but also on the consistency among identifiers from different sources (name, implementation, and documentation), and that the definition of practices that result in poor quality lexicon increases developer awareness and thus contributes to the improvement of the lexicon.

To validate our thesis we first strengthen the evidence that measures evaluating the lexicon quality are an asset for fault explanation and prediction. Then, we show that the quality of the lexicon also depends on the consistency among identifiers. To this end, we define a catalog of Linguistic Antipatterns (LAs) related to inconsistencies—among the name, implementation, and documentation (e.g., comment) of a program entity—and we show that the majority of the industrial and open-source developers that we surveyed perceived LAs as poor practices. Finally, we perform a survey with developers to understand the factors that may prevent the improvement of the lexicon, which is often performed through renaming.

In the subsequent sections we describe the contributions that allow us to validate our thesis.

1.2.1 Evidence on the Relation between Identifier Quality and Source Code Quality in terms of Fault Explanation and Prediction

Several factors contribute to the faultiness of a program entity. The structural complexity of the source code is one of the factors and it is widely studied to predict fault prone entities. Another factor that we believe contributes to the faultiness of entities is the source code lexicon. In this dissertation, we provide evidence on the relation between lexicon quality and source code quality in terms of fault explanation and prediction.

First, we define a measure, named High Entropy and High Context Coverage (HEHCC), characterizing the dispersion of terms composing source code identifiers. Entropy characterizes the *physical* dispersion of terms—i.e., in how many entities a term is used—while the context coverage characterizes the *conceptual* dispersion of a term—i.e., how unrelated are the entities in which the same term is used. We show that programming entities that contain HEHCC terms—i.e., terms used in many entities and in different contexts—are more fault prone. Using linear regression models, we also show that structural complexity metrics such as Lines Of Code (LOC) only partially explains the information captured by the newly defined lexicon metric.

We also show that adding existing LBS to structural metrics such as the Chidamber and Kemerer object-oriented metrics suite (CK) improves the prediction of fault prone classes. Using Principal Component Analysis (PCA) we compare the information captured by the

two types of metrics—lexicon and structural—and show that LBS capture new information compared to the CK metrics. Results also show that among the list of LBS, particularly poor practices are overloaded identifiers, use of synonyms, and inconsistent use of terms.

Our contributions resulted in two conference publications:

— Venera Arnaoudova, Laleh Mousavi Eshkevari, Rocco Oliveto, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "Physical and Conceptual Identifier Dispersion: Measures and Relation to Fault Proneness". In: *Proceedings of the International Conference on Software Maintenance (ICSM) - ERA Track*. 2010, pp. 1–5. **Best paper award**.

— Surafel Lemma Abebe, Venera Arnaoudova, Paolo Tonella, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "Can Lexicon Bad Smells improve fault prediction?" In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2012, pp. 235–244.

1.2.2 Definition of Linguistic Antipatterns

Software antipatterns (Brown *et al.*, 1998) are opposite to design patterns (Gamma *et al.*, 1994), i.e., they identify "poor" solutions to recurring design problems. For example, Brown's 40 antipatterns describe the most common pitfalls in the software industry (Brown *et al.*, 1998). They are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem, or misusing good solutions, i.e., design patterns.

Inspired from software antipatterns, we define a new family of software antipatterns, named Linguistic Antipatterns (LAs). LAs shift the perspective from the source code structure towards its consistency with the lexicon:

> **Linguistic Antipatterns (LAs)** in software projects are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding.

The presence of inconsistencies can be particularly harmful for developers that can make wrong assumptions about the code behavior or spend unnecessary time and effort to clarify it when understanding source code for their purposes. Therefore, highlighting the presence of LAs is essential for producing code that is easy to understand.

An example of a LA that we have named *Attribute signature and comment are opposite* occurs in class `EncodeURLTransformer` of the *Cocoon*[1] project. The class contains an attribute named `INCLUDE_NAME_DEFAULT` whose comment documents the opposite, i.e., a *"Configuration default exclude pattern"*. Whether the pattern is included or excluded is therefore unclear from the comment and name. Another example of a LA called *"Get" method does not return*

1. `http://cocoon.apache.org`

occurs in class `Compiler` of the *Eclipse*[2] project where method `getMethodBodies` is declared. Counter to what one would expect, the method neither returns a value nor clearly indicates which of the parameters will hold the result.

The definition of LAs have resulted in the following conference publication:

— Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "A New Family of Software Anti-Patterns: Linguistic Anti-Patterns". In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 2013, pp. 187–196.

1.2.3 Relevance of Linguistic Antipatterns to Developers

Although tools may detect instances of (different kinds of) bad practices, they may or may not turn out to be actual problems for developers. For example, by studying the history of projects Raţiu *et al.* (2004) showed that some instances of antipatterns, e.g., *God classes* being persistent and stable during their life, could be considered harmless. To understand whether LAs would be relevant for software developers, we must answer the following general question:

— Do developers perceive LAs as indeed poor practices?

We empirically answer the general question stated above, by conducting two different studies. In *Study I*, we showed to 30 developers an extensive set of code snippets from three open-source projects, some of which contain LAs, while others do not. Participants were *external* developers, i.e., people that have not developed the code under investigation, unaware of the notion of LAs. The rationale here is to evaluate how relevant are the inconsistencies, by involving people having no bias—neither with respect to our definition of LAs, nor with respect to the code being analyzed. In *Study II*, we involved 14 *internal* developers from 8 projects (7 open-source and 1 commercial), with the aim of understanding how they perceive LAs in projects they know, whether they would remove them, and how (if this is the case). Here, we first introduce to developers the definition of the specific LA under scrutiny, after which they provide their perception about examples of LAs detected in their project.

Results indicate that *external* and *internal* developers perceive LAs as poor practices and therefore must be avoided—69% and 51% of the participants in *Study I* and *Study II*, respectively. More important, 10% (5 out of 47) of the LAs shown to internal developers during the study have been removed in the corresponding projects after we pointed them out.

Our results are currently under review in the Journal of *Empirical Software Engineering (EMSE)*:

2. `http://www.eclipse.org`

— Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. "Linguistic Antipatterns: What They Are and How Developers Perceive Them". Submitted for review in *Empirical Software Engineering (EMSE)*.

1.2.4 Factors that may Prevent the Improvement of the Source Code Lexicon: A Study of Identifier Renaming

To understand whether developers evolve source code lexicon as they feel the need, we perform a survey on their renaming habits. The survey involves 71 developers of industrial and open-source projects. The main questions that the survey tackles are how often developers rename, whether they consider renaming as a straightforward activity, what are the factors that may prevent them from renaming, and what are the types of renamings that they consider worth recommending automatically.

Results show that renaming is a frequent activity that 39% of the participants perform from a few times per week to almost every day. Only 8% of the participants consider renaming to be straightforward. Some of the main factors that can prevent developers from renaming are insufficient domain knowledge (85% of the participants), code ownership (79%), close deadline (76%), the potential impact on other projects (52%), and the risk of introducing a bug (35%).

The results of the study have been published in *IEEE Transactions on Software Engineering (TSE)*:

— Venera Arnaoudova, Laleh Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "REPENT: Analyzing the Nature of Identifier Renamings". In: *IEEE Transactions on Software Engineering (TSE)*, 40 (5), 2014, pp. 502–532.

1.3 Organisation of the Dissertation

Figure 1.1 provides an overview of the dissertation chapters and relates them to the questions that drove this research and our major conclusions. The reminder of this dissertation is organized as follows:

Chapter 2—Background: This chapter discusses the necessary background. It summarizes the processing of the lexicon, the experiment process and particularly the planning and analysis steps, and the source code and lexicon metrics used in this dissertation.

Chapter 3—Literature Review: This chapter discusses related work relevant to this dissertation such as the quality of the lexicon and its relation with the software quality,

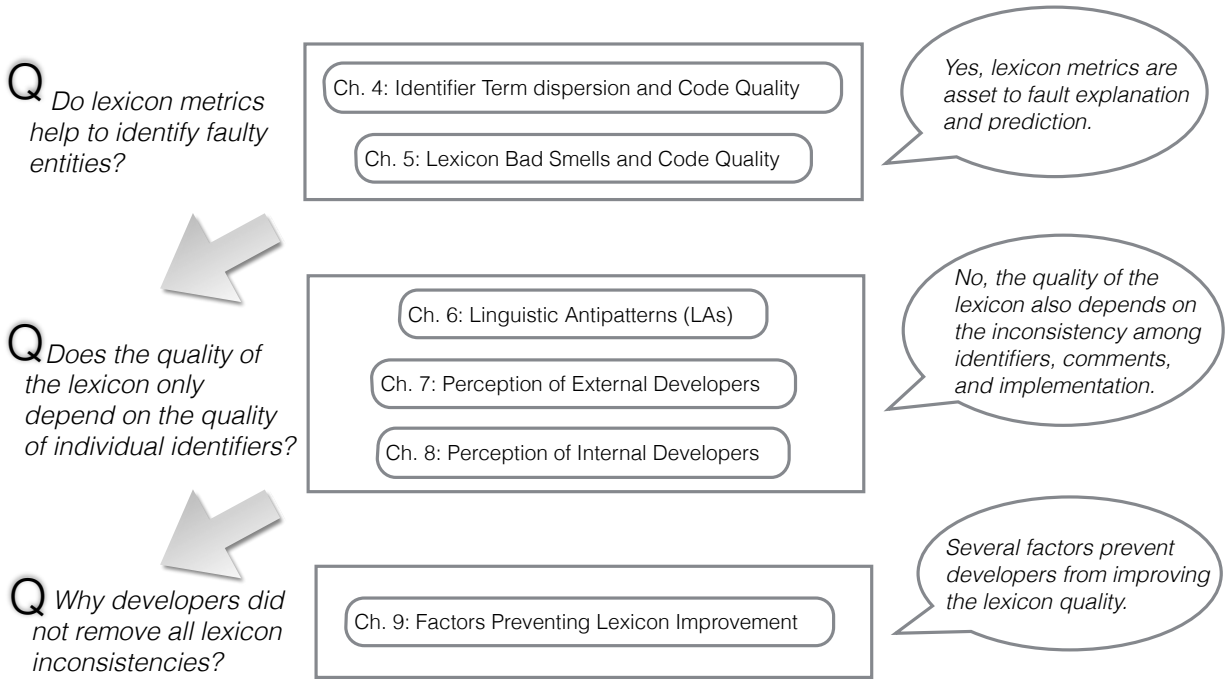


Figure 1.1 Overview of the dissertation chapters.

lexicon-related inconsistencies, and empirical studies investigating developers' perception of code smells.

Chapter 4—Identifier Term Dispersion and Code Quality: This chapter defines a new metric combining the physical and conceptual dispersion of terms. We also show that this metric, HEHCC, helps to explain software faults when combined with structural metrics of complexity such as LOC.

Chapter 5—Lexicon Bad Smells (LBS) and Code Quality: This chapters uses previously defined metrics to measure the quality of the lexicon, i.e., LBS, and investigates whether adding this information to structural metrics such as the CK metrics improves the prediction of fault prone classes.

Chapter 6—Linguistic Antipatterns (LAs): This chapter formulates the notion of source code Linguistic Antipatterns (LAs), i.e., *recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity* and defines a catalog of 17 types of LAs related to inconsistencies. We implement possible detection algorithms in a prototype tool called Linguistic AntiPattern Detector (LAPD), which we use to study the importance of the phenomenon.

Chapter 7—LAs: Perception of External Developers: This chapter reports the design and results of *Study I*, i.e., *external* developers' perception of LAs. It involves 30 developers and an extensive set of code snippets from three open-source projects, some of

which containing LAs, while others not. Participants are not familiar with the code under investigation and unaware of the notion of LAs. The rationale here is to evaluate how relevant are the inconsistencies, by involving people having no bias—neither with respect to our definition of LAs, nor with respect to the code being analyzed.

Chapter 8—LAs: Perception of Internal Developers: This chapter reports the design and results of *Study II*, i.e., *internal* developers' perception of LAs. We involve 14 developers from 8 projects (7 open-source and 1 commercial), with the aim of understanding how they perceive LAs in projects they know, whether they would remove them, how (if this is the case), and what causes LAs to occur. Here, we first introduce to developers the definition of the specific LA under scrutiny, after which they provide their perception about examples of LAs detected in their project.

Chapter 9—Factors Impacting the Improvement of the Lexicon: This chapter investigates why internal developers remove only part of the inconsistencies that we pointed them out. In particular, it reports the design and results of a survey on renaming—i.e., the typical action of resolving LAs. 71 developers of industrial and open-source projects participated in the survey. Our goal is to understand whether developers consider renaming a straightforward activity and what are the factors that may prevent them from renaming.

Chapter 10—Conclusion and Future Directions: This chapter concludes the dissertation by summarizing the contributions of our work and outlining possible future directions.

Appendix A—LAs: Detection Algorithms: This chapter provides possible detection algorithms for the catalogue of LAs defined in Chapter 6.

Appendix B—Studied Projects: This chapter provides details related to the projects studied in this dissertation.

CHAPTER 2

BACKGROUND

This section defines the necessary background for this dissertation. In particular, it defines the steps to process the source code lexicon (Section 2.1), provides details on the different steps of an experiment process (Section 2.2), and in particular the planning (Section 2.3) and analysis and interpretation (Section 2.4) steps, and summarizes the metrics that we use (Section 2.5).

2.1 Processing the Source Code Lexicon

In the subsequent paragraphs we provide details on the processing of source code lexicon. In particular, the process consists of the extraction of identifiers and comments from program entities (Section 2.1.1), splitting identifiers into terms (Section 2.1.2), POS tagging and parsing (Section 2.1.3), and establishing semantic relations among terms (Section 2.1.4).

2.1.1 Extraction

The extraction of identifiers and comments can be performed in many ways. One can process programs as if they are text documents and filter out language keywords. Such processing may be sufficient when one is not interested in knowing where identifiers come from—i.e., method name, return type, etc. Another way to extract the lexicon is to use a language specific parser that creates an Abstract Syntax Tree (AST) and to collect the identifiers and comments from the AST nodes of interest. For Java for example, this can be done using the Eclipse Java Development Tools (JDT). Finally, a third option would be to use a tool that parses source code written in several programming languages and which transforms it into an intermediate representation. One can then parse the intermediate representation without worrying about parsing a programming language. We use this last option to extract identifiers and comments from Java and C++ projects. In particular, we use the *srcml* tool (Collard *et al.*, 2003), which parses source code and produces an XML-based parse tree. We use the tree to identify the various source code elements of interest—e.g., attribute name and type, method name and parameters, etc. In addition, we can also extract from source code other pieces of information needed for our analysis, i.e., the presence of control flow or conditional statement, the usage of particular variables/parameters in conditional statements, and exception handling. When encountering comments in the source code, we attach them

to the entities that they precede. However, when a comment follows an entity declaration and it starts on the same line then we attach it to the preceding entity.

2.1.2 Splitting

This step aims at identifying term composing identifiers. Thus, for example, the identifier `getMsg` will be split into the two terms composing it: `get` and `Msg`. Note that a term can be a dictionary word, an abbreviation, or an acronym. Terms composing identifiers are glued using camelCase—i.e., each next word starts with upper case—and–or non-alphabetic characters—e.g., underscore. For Java, splitting identifiers using camelCase and underscore heuristics is largely sufficient (Madani *et al.*, 2010). However, there are several techniques that propose smarter splitting approaches (Enslen *et al.*, 2009; Lawrie et Binkley, 2011; Guerrouj *et al.*, 2013), when no clear heuristic is used to compose words—e.g., `getname`—and when the identifier contains acronyms and abbreviations, e.g., in cases such as `cntrlPrnt`.

2.1.3 POS Tagging and Parsing

In natural language, a word carries a specific meaning. Words are often grouped into phrases which in turn can be combined to form sentences. By analogy with natural language, to grasp the meaning of an identifier, one cannot rely only on the terms constituting the identifier in isolation. For example, the term `visible` (from the identifier `JavadocNotVisibleReference`) and the term `hidden` (from the identifier `JavadocHiddenReference`) have opposite meaning, whereas the identifiers have the same meaning.

Thus, after identifiers have been split into terms, we must build a sentence out of the terms, perform a POS analysis and parsing to identify that the terms `visible` and `hidden` are both adjectives modifiers specifying the term `reference` and that the term `visible` is negated (see Figures 2.1 and 2.2).

To parse identifiers and comments, we use Stanford CoreNLP (Toutanova et Manning, 2000)—a set of tools allowing to perform POS analysis and identify dependencies among words. The Stanford CoreNLP classifies terms using the Penn Treebank Tagset (Marcus *et al.*, 1993), thus not only distinguishing between nouns, verbs, adjectives, and adverbs, but also distinguishing between the different forms, e.g., plural noun, verb past participle, etc.

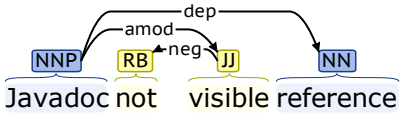


Figure 2.1 Parsing the sentence "Javadoc not visible reference" using Stanford CoreNLP.

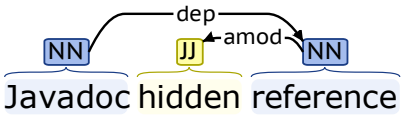


Figure 2.2 Parsing the sentence "Javadoc hidden reference" using Stanford CoreNLP.

As identifiers do not always follow well-formed grammatical structure, before applying POS analysis using natural language tools we apply a sentence template. Different templates have been proposed in the literature (Abebe et Tonella, 2010; Binkley *et al.*, 2011). Binkley *et al.* (2011) evaluate different templates and provide evidence that the List Item Template template outperforms the other three templates they evaluated. Thus, for the identifier `inclusionPatterns` the template produces *inclusion patterns*. However, if the first term is a verb, as it is suggested according to Java standard for method names, we use a different template, i.e., the verb template: "Try to <identifier terms>". Note that a template is just an aid provided to the POS tagger to guide its analysis. Thus, for the method name `parse`, after applying the verb template we obtain the sentence *Try to parse* which we then pass to the Stanford CoreNLP tool.

2.1.4 Analyzing Semantics

To find semantic relations between terms, one can use a domain specific ontology or a general ontology. Domain specific ontologies require the time and effort of experts and thus are not always available. General ontologies such as WordNet (Miller, 1995) exist for English and can be used instead. Words in WordNet are organized based on their relations. Synonyms are grouped into unordered sets, called synsets, which in turn are related using semantic and lexical relations. Thus, using WordNet, we are able to identify semantic relations among terms such as synonymy, antonymy, hyponymy, hypernymy, etc.

Recently, two approaches have been proposed towards automatically constructing domain ontologies from software projects. For example, in their recent work Yang et Tan (2013) propose an approach to mine semantically related words in a project or multiple projects from the same domain. Similar work has been done by Howard *et al.* (2013) where the authors mine semantically similar words across projects from multiple domains.

2.2 Experiment Process

Conducting an experiment involves several main steps (Wohlin *et al.*, 2000):

Definition: In this step, the objectives and the goal of the experiment must be defined.

Planning: During the planning, one must define the context of the experiment, select the variables to study, select the subjects and objects, choose the design of the experiment, and identify the threats to the validity.

Operation: This step consists of preparing the material, running the experiment, and collecting the data.

Analysis and interpretation: In this step, the collected data is analyzed (using descriptive statistics and data visualization techniques), possibly reduced, the statistical test are performed, and the results are interpreted.

Presentation and package: Finally, the experiment is documented.

We provide more background on the planning and on the analysis and interpretation steps in Sections 2.3 and 2.4, respectively.

2.3 Experiment Planning

The experiment planning is a fundamental step in the experiment process. In particular, in this step, the experiment hypotheses are formally stated (Section 2.3.1), the variables are selected (Section 2.3.2), and the threats to the validity are identified (Section 2.3.3).

2.3.1 Hypothesis Testing

Hypothesis testing is the basis of statistical analysis. During hypothesis testing, two hypotheses must be formally stated as follows:

Null hypothesis (H_0) States that there is no difference in the trends of two populations. This is the hypothesis that one wants to reject.

Alternative hypothesis (H_1) Is accepted when the null hypothesis is rejected. Accepting the alternative hypothesis means that the difference between two populations is not coincidental.

Hypothesis testing involves two types of errors:

Type-I-error (α) The error of rejecting H_0 when it is true.

Type-I-error (β) The error failing to reject H_0 when it is false.

Whether H_0 is rejected or not depends on 1) the confidence of the rejection provided the experimental data and 2) the minimal confidence required. Typically, in software engineering a minimal confidence required is 95%, which means accepting maximum 5% Type-I-error—i.e., $\alpha = 0.05$. If the confidence of the rejection provided the experimental data is sufficient—i.e., $p - value \leq \alpha$—then H_0 is rejected in favour of H_1. If H_0 cannot be rejected, nothing

can be said about the two populations. However, when H_0 is rejected, the confidence of the rejection is $1 - p - value$. In this dissertation we consider $\alpha = 0.05$.

2.3.2 Variables

Variable selection is an important step in the experiment planning. There are two kinds of variables: *dependent* and *independent* variables. A dependent variable, also called response variable, is the variable that we are interested to study, i.e., we observe how it varies when the independent variables change. The independent variables are the input variables that are manipulated and controlled. The independent variables that vary to evaluate the impact on the dependent variables are called factors, or exposure variables. The rest of the independent variables that we account for are called control variables. Finally, a confounding variable is a variable that changes systematically when one or more independent variables change, thus providing an alternative explanation of the observed relation between the independent and dependent variables.

2.3.3 Threats to Validity

Threats to validity deal with doubts that question the quality and accuracy of an experimentation. A common classification involves four categories, namely threats to *conclusion*, *internal*, *construct*, and *external validities* (Wohlin *et al.*, 2000). For case studies, Yin (1994) also discusses threats to *reliability validity*.

Threats to *conclusion validity*, sometimes referred to as statistical conclusion validity (Cook et Campbell, 1979), concern the relation between the independent and dependent variables and factors that may prevent us from drawing the correct conclusions. Examples of threats to conclusion validity are the power of a statistical test (e.g., it may not be high enough to allow us to reject the null hypothesis); violated assumptions of the statistical test (e.g., perform a parametric test when the data is not normally distributed); reliability of the measures.

Threats to *internal validity* concern the relation between the independent and dependent variables and factors that could have influenced the relation with respect to the causality. Examples include confounding factors such as subjects becoming tired or motivated as time passes, diffusion of information among different groups of subjects, etc.

Threats to *construct validity* concern the relation between theory and observation, and they are mainly related to the design of the experiment and social factors. Threats to construct validity of the design include the mono-operation bias, e.g., when a single subject is considered. Social threats to construct validity include possible bias related to the experimenter and–or the subjects such as the experimenter bias and hypothesis guessing.

Threats to *external validity* concern the generalizability of the findings outside the experimental settings. Examples of threats to external validity include the selection of subjects and–or objects that are not representative of the studied population.

Threats to *reliability validity* concern the ability to replicate a study with the same data and to obtain the same results. Examples of threats to reliability validity include insufficient details about the data collection and the analysis procedures.

2.4 Experiment Analysis and Interpretation

In the analysis and interpretation step of the experiment process, the data is often visualized (Section 2.4.1) before any analysis. In Section 2.4.2, we provide background on the prediction models used to model the dependent variable and in Section 2.4.3 we present the measures used to evaluate those models. We also describe other measures for dependencies (Section 2.4.4) and discuss confidence level and interval (Section 2.4.5).

2.4.1 Data Visualization

Box Plot

Boxplots are used to visualize the central tendency and dispersion of the data. Figure 2.3 depicts an example of a boxplot. A box is drawn to delimit the lower and upper quartiles (i.e., 25% and 75% percentiles, respectively); the thick line in the box is the median. The tails of the box represent the theoretical bounds of all data points provided that they have a normal distribution.

Violin Plot

Violin plots (Hintze et Nelson, 1998) combine boxplots and kernel density functions, thus providing a better indication of the shape of the distribution. Figure 2.4 shows two examples of violin plots that if plotted with a boxplot produce the same result, i.e., both violin plots correspond to Figure 2.3. Violin plots however, allow to observe a bimodal distribution of the data on the left example shown in Figure 2.4. The dot inside the violin plot represents the median; a thick line is drawn between the lower and upper quartiles; a thin line is drawn between the lower and upper tails.

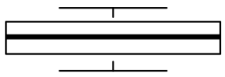


Figure 2.3 Example of a boxplot.

Figure 2.4 Examples of violin plots.

2.4.2 Prediction Models

Prediction models are used to model the relation between the dependent variable and the independent variable(s).

Linear Regression

A linear regression model is used when there is a linear relation between a numeric dependent variable Y and independent variables X_i. In its simplest version, it is given by the formula:

$$Y = \alpha + \beta X + \epsilon$$

where ϵ is a random error in Y independent of X. α is the intercept and β is the slope of a straight line that fits the data.

Logistic Regression

The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}{1 + e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}$$

where X_i are the independent variables, and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In a logistic regression model, the dependent variable π is commonly a dichotomous variable, and thus, assumes only two values $\{0, 1\}$—e.g., it states whether an entity is faulty (1) or not (0). The closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the entity contains a fault. A threshold is used in practice to decide on the cutoff point. In this thesis, we use 0.5. The C_i are the estimated regression coefficients, the higher the absolute value, the higher the contribution of the corresponding independent variable.

When performing multivariate regression (i.e., regression using more than one independent variables) we must account for possible risk of multicollinearity (i.e., interaction among the independent variables). A common way to deal with multicollinearity is to compute the Variance Inflation Factors (VIF) for each independent variable in the regression model and retain only those with low values—e.g., <=2.5 (Cataldo *et al.*, 2009; Shihab *et al.*, 2010).

Random Forest

Random Forest (Breiman, 2001) averages the predictions of a number of tree predictors where each tree is fully grown and is based on independently sampled values. The large number of trees avoids overfitting. Random Forest is known to be robust to noise and to

correlated variables. In Chapter 5 we use the function randomForest (package randomForest from the R environment [1]) with the number of trees being 500 (Weyuker *et al.*, 2010).

Support Vector Machine

SVM is a machine learning technique that tries to maximize the margin of the hyperplane separating different classifications. Some of the advantages of SVM include the possibility to model linear and non-linear relations between variables and its robustness to outliers.

The formula for the Gaussian Radial Basis kernel function is:

$$k(x_i, x_j) = exp(-\gamma * |x_i - x_j|^2)$$

where x_i and x_j are two data points, and γ is a parameter to be estimated.

We used the Support Vector Machine model (package e1071 from the R environment) svm (kernel="radial"). This kernel showed good performance in previous works (Elish et Elish, 2008).

2.4.3 Performance Measures

In the literature, various metrics are used to evaluate the prediction capability of independent variables and to compare prediction models (Zhou et Leung, 2006; Zimmermann *et al.*, 2007; Mende et Koschke, 2009; Hassan, 2009; Weyuker *et al.*, 2010). We have categorized these metrics into three groups: *rank*, *classification*, and *error* metrics. Below we present the details of each category in the context of fault proneness of program entities.

Rank

Rank metrics sort the program entities based on the value of the dependent variable assigned to each entity. Then a cumulative measure is computed using the actual values of the dependent variable over the ranked entities to assess the model and–or the independent variables. In our study, we consider two types of rank metrics: P_{opt} and Fault Percentile Average (FPA).

P_{opt} is an extension of the Cost Effective (CE) measure defined by Arisholm *et al.* (2007). P_{opt} takes into account the costs associated with testing or reviewing an entity and the actual distribution of faults, by benchmarking against a theoretically possible optimal model (Mende et Koschke, 2009). It is calculated as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the optimal

1. http://www.r-project.org/

and the predicted cumulative lift charts. The cumulative lift chart of the *optimal* curve is built using the actual defect density of entities sorted in decreasing order of the defect density (and increasing lines of code, in case of ties). The cumulative lift chart of the *predicted* curve is built like the optimal curve, but with entities sorted in decreasing order of fault prediction score.

FPA is obtained from the percentage of faults contained in the top $m\%$ of entities predicted to be faulty. It is defined as the average, over all values of m, of such percentage (Weyuker *et al.*, 2010; Bell *et al.*, 2011). On entities listed in increasing order of predicted numbers of faults, FPA is computed as:

$$\frac{1}{NK}\sum_{k=1}^{K}(k * n_k)$$

where N is total number of actual faults in a project containing K entities, n_k is the actual number of faults in the entity ranked k (Weyuker *et al.*, 2010).

In Chapter 5, we predict the probability of fault proneness of an entity instead of the number of faults. Hence, we have adapted the metrics by using the predicted probability of fault proneness to sort the entities, and 0 and 1 are used as a replacement of the number of defects: 1 is used when an entity is actually faulty; 0 otherwise.

Classification

Predicting fault proneness of an entity is a classification problem. Hence, in various studies the confusion matrix (shown in Table 2.1) is used as the basis for the evaluation of the models and analyze the prediction capability of the independent variables. True Positives (TP) and True Negatives (TN) are correct predictions; False Positives (FP) and False Negatives (FN) are incorrect predictions. The following measures are computed using the confusion matrix:

Table 2.1 Confusion matrix (TP=True Positives, TN=True Negatives, FP=False Positives, FN=False Negatives).

	Actual faulty	**Actual not faulty**
Predicted faulty	TP	FP
Predicted not faulty	FN	TN

Accuracy (A) measures how accurately entities are classified as faulty and non-faulty by the predictor. It is computed as the ratio of the number of entities that are correctly predicted as faulty and non-faulty to the total number of entities:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

A score of 1 indicates that the model used for the prediction has classified all entities as faulty and non-faulty correctly.

Precision (P) (sometimes referred to as correctness) indicates how well a predictor identifies the faulty entities as faulty. It is computed as the ratio of the correctly predicted faulty entities to the total number of predicted faulty entities:

$$P = \frac{TP}{TP + FP}$$

A prediction model is considered very precise if all entities predicted as faulty are actually faulty, i.e., if $P = 1$.

Recall (R) (sometimes referred to as completeness) indicates how many of the actually faulty entities are predicted as faulty. Recall is computed as the ratio of the number of correctly predicted faulty entities to the total number of actually faulty entities:

$$R = \frac{TP}{TP + FN}$$

F-measure (F) is a measure used to combine the above two inversely related classification metrics—i.e., precision and recall—and it is computed as their harmonic mean:

$$F = \frac{2 * P * R}{P + R}$$

Matthew's Correlation Coefficient (MCC) is a measure commonly used in the bioinformatics community to evaluate the quality of a classifier (Matthews, 1975). It is a quite robust measure in the presence of unbalanced data. MCC is computed as:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The value of MCC ranges from -1 to 1. -1 indicates a complete disagreement while 1 indicates the opposite.

Error

In the last category of the evaluation metric types, we have *absolute error (E)*.

Absolute Error (E) is a measure based on the number of faults incorrectly predicted or missed:

$$E = \sum_{k=1}^{K} |\hat{y_k} - y_k|^2$$

where $\hat{y_k}$ is the predicted number of faults in entity k and y_k the actual number of faults (Hassan, 2009). In Chapter 5, we are interested in the fault proneness of an entity and not in the number of faults it contains. Thus, we use 0 and 1, as a replacement of the number of faults. 1 is used when an entity is actually faulty/predicted to be faulty and 0 otherwise. Unlike the other evaluation metrics, for absolute error a value closer to 0 indicates better prediction capability.

2.4.4 Dependency and Effect Size Measures

In this section we summarize techniques and measures of the dependency between variables and the strength of the relation, if any.

PCA

Principal Component Analysis (PCA) is a technique that uses solutions from linear algebra to project a set of possibly correlated variables into a space of orthogonal Principal Components (PC), or eigen vectors, where each PC is a linear combination of the original variables. PCA is used to reveal hidden patterns that cannot be seen in the original space and to reduce the number of dimensions. When using PCA it is a common practice to select a subset of the principal components and discard those that explain only a small percentage of the variance. For each principal component, PCA reports the coefficients of the attributes on the corresponding eigen vector. Those coefficients are interpreted as the importance of the attribute on the PC.

Pearson's Product-moment Correlation

Pearson's Correlation coefficient (r) is a parametric statistical test measuring dependency between two variables. The r-value is between -1 and $+1$. If there is no correlation then r would be 0. However, when r is 0 this only means that there is no linear correlation between the two variables. The r coefficient requires the data to be normally distributed.

Spearman's Correlation

Spearman's correlation coefficient (r_s) is a non-parametric statistical test for dependency between two variables. It is calculated using the ranks of the data rather than the actual values. r_s varies between -1 and +1. Spearman's correlation is preferred for software design metrics over the Pearson's correlation as we are often dealing with skewed data (Briand *et al.*, 2000).

Prop Test

Prop Test (or Pearson's chi-squared test) is a non-parametric statistical test used to determine if there exist a relation between two variables by comparing proportions. The input is provided as a contingency table and the values in each cell are assumed to be greater than 5 (Sheskin, 2007).

Fisher's Exact Test

Fisher's Exact Test is a non-parametric statistical test used to determine whether two categorical variables are independent by comparing their proportions. It is often used to replace the Pearson's chi-squared test for small samples.

Mann-Whitney Test

The Mann-Whitney (Wohlin *et al.*, 2000)—also known as the two-sample Wilcoxon test—is a non-parametric test used as an alternative to the two-sample t-test when the data is not normally distributed. Given two samples, Mann-Whitney tests whether they come from the same distribution (i.e., H_0) based on their ranks rather than the data itself.

Odd Ratio (OR)

Odd Ratio (OR) measure the strength of the relation between variables. Consider two binary random variables X and Y as shown in Table 2.2, where each cell represent the number of observations for the respective values of X and Y. The odd ratio is defined as:

$$OR = \frac{n_{11} * n_{00}}{n_{10} * n_{01}}$$

An $OR = 1$ means that the distribution of Y over X is equal. $OR \geq 1$ ($OR \leq 1$) means that observations where Y=1 have higher (lower) chances to have X=1. For example, if Y is the dependent variable measuring the fault proneness of a program entity and X is a boolean

variable indicating whether the entity has poor lexicon, an $OR = 2$ would mean that entities with poor lexicon have two times higher chances to be fault prone.

Cliff's Delta

Cliff's delta (d) effect size (Grissom et Kim, 2005)—also known as the dominance measure—is a non-parametric statistic estimating whether the probability that a randomly chosen value from a group (e.g., group 1) is higher than a randomly chosen value from another group (e.g., group 2), minus the reverse probability. The value belongs to the range $[-1, 1]$ and are interpreted as follows: When $d = 1$ there is no overlap between the two groups and all values from group 1 are greater than the values from group 2. When $d = -1$ there is again no overlap between the two groups but all values from group 1 are lower than the values from group 2. When $d = 0$ there is a complete overlap between the two groups and thus there is no effect size, i.e., the chances that values from group 1 are greater than values from group 2 are equal to the chances the values being lower. Between 0 and 1 the magnitude of the effect size is interpreted as follows: $0 \leq |d| < 0.147$: negligible, $0.147 \leq |d| < 0.33$: small, $0.33 \leq |d| < 0.474$: medium, $0.474 \leq |d| \leq 1$: large.

2.4.5 Confidence Level and Confidence Interval

Confidence interval is defined as plus and minus a confidence value around the result. The confidence level provides an indication of certainty. For example, if an approach exhibits a precision of 55% with a confidence value of 5 and 99% confidence level, this means that we are 99% sure that the true percentage is between 50% (i.e., $55 - 5$) and 60% (i.e., $55 + 5$).

2.5 Metrics

A metric is a mapping between an attribute of an object and a value. During the experiment planning step, when variables are selected, the metrics used to measure the variables must also be selected.

2.5.1 Entropy

Shannon (Cover et Thomas, 2006) measures the amount of uncertainty, or entropy, of a discrete random variable X as:

$$H(X) = -\sum_{x \in \varkappa} p(x) \cdot log(p(x))$$

Table 2.2 Contingency table for two binary random variables X and Y.

	Y=1	Y=0
X=1	n_{11}	n_{10}
X=0	n_{01}	n_{00}

where $p(x)$ is the mass probability distribution of the discrete random variable X and $\varkappa$ is its domain. For $\varkappa$ containing only two possible values, i.e., $\varkappa = \{x_1, x_2\}$, entropy is maximized (i.e., $H(X) = 1$) when the uncertainty is largest—i.e., for $p(x_1) = p(x_2) = 0.5$—and minimized (i.e., $H(X) = 0$) when there is absolute certainty—i.e., for $p(x_1) = 1$ and $p(x_2) = 0$ or vice versa (i.e., $p(x_1) = 0$ and $p(x_2) = 1$).

2.5.2 Structural Code Metrics

Table 2.3 shows the structural metrics used in this dissertation. The list consists of the set of well-known CK metrics (Chidamber et Kemerer, 1994), two metrics measuring the lack of cohesion in methods (LCOM2 and LCOM5) defined by Briand *et al.* (1998), and two metrics counting the number of declared attributes and methods (Lorenz et Kidd, 1994).

Table 2.3 List of considered structural metrics.

Acronym	**Description**
CBO (Chidamber et Kemerer, 1994)	Coupling between objects
DIT (Chidamber et Kemerer, 1994)	Depth of Inheritance Tree
LCOM1 (Chidamber et Kemerer, 1994)	Lack of COhesion in Methods 1
LCOM2 (Briand *et al.*, 1998)	Lack of COhesion in Methods 2
LCOM5 (Briand *et al.*, 1998)	Lack of COhesion in Methods 5
LOC (Chidamber et Kemerer, 1994)	Line Of Code
NAD (Lorenz et Kidd, 1994)	Number of Attributes Declared
NMD (Lorenz et Kidd, 1994)	Number of Methods Declared
NOC (Chidamber et Kemerer, 1994)	Number Of Children
RFC (Chidamber et Kemerer, 1994)	Response For a Class
WMC (Chidamber et Kemerer, 1994)	Weighted Methods per Class

2.5.3 Lexicon Bad Smells (LBS)

Abebe *et al.* (2009b) define a catalog [2] of Lexicon Bad Smells (LBS) (Abebe *et al.*, 2009b) as anomalies that reduce the quality of identifier names. Below we present a summary of the LBS and illustrate them with an example.

2. The catalog is available online: `http://selab.fbk.eu/LexiconBadSmellWiki/`

Extreme contraction refers to extremely short terms used in identifiers due to an excessive word contraction, abbreviation, or acronym. An example of such identifier is `aSz` (`a`=array, `sz`=size). This rule does not apply to prefixes introduced due to the naming conventions adopted in the project (e.g., `m_` is a prefix used in the Hungarian notation to mark attributes of a class), common programming and domain terms (e.g., `msg`, `SQL`, etc.), and short dictionary words (e.g., `on`, `it`, etc.).

Inconsistent identifier use refers to two or more identifiers that refer to a concept in an inconsistent way. Operationally, an identifier is considered inconsistent when it is contained in another identifier of the same type (e.g., another class/method/attribute name), which is found in the same container entity (e.g., package, class). In the example shown in Figure 2.5, the attribute `path` is inconsistent as it is contained in two other identifiers `absolute_path` and `relative_path` of the same type (i.e., also attributes) and defined in the same container (i.e., class `Documents`).

```
class Documents {
   private String absolute_path;
   private String relative_path;
   private String path;//path is inconsistent
}
```

Figure 2.5 Example of *inconsistent identifier use*.

Meaningless terms refers to metasyntactic identifier names like `foo` and `bar`.

Misspelling refers to misspelled words in an identifier.

Odd grammatical structure refers to identifiers constructed using inappropriate grammatical structure for the specific kind of program entity they represents (e.g., a class name contains a verb, method names do not start with a verb, etc.). Figure 2.6 shows an example of class and method identifier names that are grammatically incorrect.

```
class Compute //compute is a verb {
   public void addition();//addition is a noun
}
```

Figure 2.6 Example of *odd grammatical structure*.

Overloaded identifiers refers to identifiers that include more than one semantics and hence multiple responsibilities of the respective program entities that they represent (e.g., a method name contains two verbs). The method name `create_export_list()`, for example, could refer to two tasks: creating and exporting a list.

Useless type indication refers to identifiers that provide redundant information about their type. For example, the attribute name `nameString` in the attribute declaration `String nameString` gives redundant information about its type. This rule does not apply for a static attribute used to realize the singleton design pattern, which usually has the same name as the class, and individual characters or groups of characters used to denote the type of the variable, if these are prescribed in the adopted naming conventions (i.e., in the Hungarian notation, `i` is a prefix used in identifiers of integer type).

Whole-part refers to a term used to name a concept that appears also in the name of its properties or operations. Figure 2.7 shows the ambiguous and redundant use of the concept `account`. Exceptions to this rule are a static attribute, used to realize the singleton design pattern and constructor methods, as they have the same name as the class.

```
class Account {
   int account; //Ambiguous use
   void computeAccount();
   //Account is redundant information
}
```

Figure 2.7 Example of *whole-part*.

Synonyms and similar identifiers refers to synonym or similar terms used to construct the identifiers representing different entities declared in the same container, such that differentiating between their responsibilities is difficult. An example of this LBS is the use of the synonym terms `copy` and `replica` in identifiers `idCopy` and `idReplica`.

Terms in wrong context refers to using terms that pertain to the domain of another container (e.g., package). This indicates that the entity named by such terms may be misplaced. For example, in Figure 2.8 the class `TypeDetector` is wrongly placed in package `collections` or incorrectly named as all the other classes that refer to detector are in package `detectors`.

No hyponymy/hypernymy in class hierarchies refers to an identifier representing a child class in an inheritance hierarchy but is not hyponym of the identifier of its parent

```
package collections;
class IntArray;
class TypeDetector ;
package detectors;
class MuonDetector ;
class PhosDetector ;
class HLTDetector ;
```

Figure 2.8 Example of *terms in wrong context*.

class. An example of such LBS is a class named `Violin` that extends the class `Mammal`. This violation is hard to assess when a class identifier is constructed from more than one term or contains abbreviations, contractions, or acronyms.

Identifier construction rules refers to identifiers that do not follow a standard naming convention adopted in the project, prescribing the use of proper prefixes, suffixes, and term separators. In a project that adopts the Hungarian notation, for example, an attribute that does not start with one of the prefixes defined for the attributes (e.g., `m_`) is considered to have this LBS.

CHAPTER 3

LITERATURE REVIEW

This chapter discusses related work, concerning the importance of source code lexicon (Section 3.1), the relationship between source code lexicon and software quality (Section 3.2), the identification and analysis of lexicon-related inconsistencies (Section 3.3), and empirical studies aimed at investigating developers' perception of code smells (Section 3.4).

3.1 Importance of Source Code Lexicon

Takang *et al.* (1996) empirically investigate the effects of comments and use of abbreviations in program identifiers on program comprehension. Results show that commented programs and programs containing full words identifiers are easier to understand.

Relf (2005) proposes a tool to dynamically assist developers in the choice of identifiers, i.e., as part of their source code editor. The author performed an experiment to understand whether developers (professionals and students) improve the quality of their identifiers when they are provided with such dynamic feedback. The quality of identifiers is measured with 19 identifier-naming style guidelines such as the length of the identifier, the number of words, whether is composed of English words etc. Results show that participants who used the dynamic feedback produced identifiers with a statistically significant higher quality than participants who did not have the tool.

Lawrie *et al.* (2006b, 2007b) empirically assess the quality of source code identifiers—considering identifiers with full-words, single-letters, and abbreviation—and their impact on program comprehension and developers' short term memory. They show that better comprehension is achieved when full word identifiers are used rather than single letter identifiers. They also show that in many cases abbreviations are as useful as the full word identifiers, especially for women (Lawrie *et al.*, 2006b). When considered in the light of limited human short-term memory, well-chosen abbreviations may be preferable in some situations since identifiers with fewer syllables are easier to remember (Lawrie *et al.*, 2007b).

Lawrie *et al.* (2007a) define identifier quality as the use of natural language words, coherent abbreviations, and common library identifiers. Their empirical investigation over open-source and proprietary projects show that modern programs contain higher quality identifiers. They also observe a difference between open-source and proprietary projects—open-source projects include higher percentage of dictionary words while proprietary projects include more abbre-

viations. Finally, they note that across programming languages, projects written in Java include the highest percentage of dictionary words.

Binkley *et al.* (2009) perform an empirical study of the impact of identifier style on code readability. Participants are first shown a sentence, then, in a subsequent screen they have to choose the corresponding identifier. The authors are interested in the time and accuracy of participants to find the identifiers written in different styles, namely, underscore and camel case. Results show that the use of camel case leads to more accurate results. As for time, they show that participants with no training need more time to find identifiers written in camel case style but trained participants are faster in recognizing camel case style identifiers.

Abebe *et al.* (2009b) define a catalogue of LBS—i.e., anomalies in the construction of source code identifiers—and their detection algorithms. Examples of LBS are extremely short identifiers, identifiers using synonyms or similar words, identifiers with inappropriate grammatical structure, and misspelled identifiers. Abebe *et al.* (2011) show that such LBS negatively impacts IR-based concept location.

Buse et Weimer (2010) proposed to approximate human perception of code readability (i.e., how easy a program is to understand) using different code features—such as number of characters, length of variable names, or number of comments. The automated readability measure that is built on those features is 80 percent effective in predicting readability judgements. They find that the average line length and the average number of identifiers per line are very important factors to readability. They also find that comments are moderately correlated with the annotators' notion of readability. However, the average identifier length is not, in itself, a very predictive factor. They also show that more readable code is correlated with fewer errors.

Discussion

We share with previous work the conjecture about the importance of the source code lexicon. We contribute to the existing body of research by providing empirical evidence that poor quality lexicon is correlated with fault proneness (Chapters 4 and 5). We also define a new family of poor practices related to the inconsistency of the lexicon (Chapter 6) and evaluate them by industrial and open-source developers (Chapters 7 and 8).

3.2 Lexicon Quality and Code Quality

Butler *et al.* (2009) analyze the relation between naming conventions and code quality. They evaluate the quality of identifiers in eight open-source Java libraries using 12 naming conventions. They show that there exists a statistically significant relation between flawed

identifiers (i.e., violating at least one convention) and code quality. Code quality is measured in terms of the anomalies reported by FindBugs (Hovemeyer et Pugh, 2004)—a static analysis tool that reports bug patterns.

Butler *et al.* (2010) study the relation between the quality of identifiers and the quality of the source code. Identifier quality is measured with naming conventions such as capitalisation anomalies, the length of identifiers, the use of dictionary use, etc. The authors measure code quality considering different factors, namely, cyclomatic complexity (McCabe, 1976), maintainability index (Welker *et al.*, 1997), code readability (Buse et Weimer, 2008), and anomalies as reported by FindBugs. Results show that poor quality identifier names are strongly associated with more complex, less readable, and less maintainable source code.

Buse et Weimer (2010) seek to understand if there exist a correlation between the readability metric that they define and software faults. To this end, the authors analyse the correlation between the readability metric with i) defects reported by FindBugs (Hovemeyer et Pugh, 2004), ii) code churn, and 3) defects mined from messages in the projects' Version Control System (VCS). Overall, they show that more readable code is correlated with fewer faults and is less likely to change.

Poshyvanyk et Marcus (2006) and Marcus *et al.* (2008) show that the conceptual measures of coupling and cohesion (Cocc and C3) capture new dimensions not captured by the corresponding families of structural metrics, i.e., structural coupling and cohesion respectively.

Binkley *et al.* (2007) predict the number of faults with QALP (Lawrie *et al.*, 2006c), LOC, and Source Lines Of Code (SLOC). QALP measures the similarity between method's code and comment.

Discussion

We share with the above works the conjecture that linguistic information is important and that its quality correlates with the quality of the code. Our work mainly differs from those previous works by showing that linguistic information captures a new dimension with respect to other types of metrics such as structural metrics. Thus, our work is closer to the works of Poshyvanyk et Marcus (2006), Marcus *et al.* (2008), and Binkley *et al.* (2007). The most notable difference between our approach and those works is the metrics used to capture linguistic information, i.e., we measure linguistic information with HEHCC and LBS. We evaluate the additional information that HEHCC and LBS bring compared to LOC and the CK (Chapters 4 and 5, respectively).

3.3 Lexicon Inconsistencies

Deißenböck et Pizka (2005) provide guidelines for the production of high-quality identifiers. Using bijective mappings between concepts and identifiers, they define rules for concise and consistent naming. Thus, the use of homonyms and synonyms leads to inconsistent identifiers as one identifier represents more than one concepts (homonymy) or multiple identifiers represent the same concept (synonymy). An identifier is considered concise if it corresponds to the most specific concept in the set of domain concepts. To support developers to follow these rules, Deißenböck and Pizka propose a tool supported IDENTIFIER DICTIONARY (IDD). Such a dictionary contains the lists of identifiers that are used as well as additional information such as the type associated with it and a description provided by the developers. One is then able to browse the existing identifiers when searching for a name. The tool also reports two inconsistencies when i) identical identifiers with different type exist in the project and ii) when identifiers are declared but never used in the project.

Lawrie *et al.* (2006a) extend the work of Deißenböck and Pizka by proposing to automatically detect violations of a restricted form of the synonym consistency and conciseness. They define syntactic-synonym consistency and the syntactic conciseness based on identifier containment thus, handling situations where bijective mappings between concepts and identifiers are not available.

Lawrie *et al.* (2006c) propose an IR based measure called QALP (Quality Assessment using Language Processing) to assess software quality by the degree of correspondence between source code comments and code. The correspondence is measured using the cosine similarity between, on the one hand, the terms extracted from the function's source code identifiers and, on the other hand, the function' leading and inline comments. Results show that functions with higher cosine similarities receive higher human quality assessments.

Fluri *et al.* (2007) study whether source code and comments co-evolve. They find that this is rarely the case. Particularly, the proportion of Javadoc changes that were induced by either declaration changes or method body changes is less than 50%.

Ibrahim *et al.* (2012) study the relation between comment update practices and faults. They find that not all inconsistent changes correlate with faults and that the most risky comment update practices are those where the update practice suddenly changes, e.g., when a method and its comment are updated inconsistently whereas they are usually undated consistently (and vice versa).

Some of the LBS defined by (Abebe *et al.*, 2009b) are defined as inconsistencies among identifiers. Those are *inconsistent identifier use* (similar to the syntactic-synonym inconsistency defined by Lawrie *et al.* (2006a)), *whole-part* (name a concept that appears also in

the name of its properties or operation), *synonyms and similar identifiers* (use of different terms to represent the same concept), *terms in wrong context* (terms more frequently used in different entities/subsystems), and *no hyponymy/hypernymy in class hierarchies* (the terms used in the super class are not a generalization of the terms used in the subclass).

Høst et Østvold (2009) identify naming bugs by mining inconsistencies between the method' name and its semantics. The authors characterise method semantics by mining data flow and control flow properties, e.g., read/write fields, object creation, and return a value. When analyzing method names, the authors abstract phrases generalizing method names. Rules for good names are derived for each prevalent phrase in the method corpus. They used a corpus of 100 Java projects to identify rules—common semantics—for methods with similar names. Naming bugs are identified for methods that break those rules. The detection is available via the Lancelot Eclipse plugin (Karlsen *et al.*, 2012).

Lawrie *et al.* (2010) propose GenTest to normalize the vocabulary, i.e., to align the vocabulary used in the source code with the one used in other software artifacts. Normalization consists of splitting an identifier into its constituent parts and expanding each part into a dictionary word. GenTest generates and tests all possible splits. Splits are scored using metrics from three categories, namely those incorporating soft-word characteristics, external, and internal information. When used only for splitting, the one with the highest score is selected. When used for normalization (by Normalize), a ranked list of high-scoring splits is used to prioritize the expansions considered. Later, Lawrie et Binkley (2011) further improved Normalize using a machine translation technique, namely, the maximum co-occurrence model (Gao *et al.*, 2002). The idea is to guide the expansion using global and local word co-occurrence.

Abebe et Tonella (2013) use the ontology extracted from source code (Abebe et Tonella, 2011) to help developers in choosing identifiers consistent with the concepts already used in the project. To this aims, given partially written identifiers, they suggest and rank candidate completions and replacements.

De Lucia *et al.* (2011) proposed an approach and tool—named COCONUT—to ensure the consistency between the lexicon of high-level artifacts and of source code. In their approach, the inconsistent lexicon is measured in terms of textual similarity between high-level artifacts traced to the code and the code itself. In addition, COCONUT uses the lexicon of high-level artifacts to suggest appropriate identifiers.

Tan *et al.* (2007, 2011, 2012) proposed several approaches to detect inconsistencies between code and comments combining natural language processing tools and static or dynamic program analysis. Specifically, @iComment (Tan *et al.*, 2007) detects lock- and call-related inconsistencies. @iComment first extracts assumptions in terms of rules from the method

comments. Then, using flow-sensitive and context-sensitive program analysis @iComment searches for violations of the extracted rules. The validation made by developers confirmed 19 of the detected inconsistencies. @aComment (Tan *et al.*, 2011) detects synchronization inconsistencies related to interrupt context. @aComment extracts interrupt related formal annotations, i.e., preconditions and postconditions, from both source code and comments written in natural language. The source code analysis is similar to the one performed for @iComment. The authors evaluated @aComment on the Linux Kernel, and the evaluation by developers confirmed 7 previously unknown bugs. Both @iComment and @aComment are applicable to C/C++ code. To detect inconsistencies between *Javadoc* and implementation, @tComment infers properties form *Javadoc* related to null values and exceptions; then, using dynamic analysis—random test generation—@tComment searches for violations of the inferred properties. Also in this case, Tan *et al.* reported the detected inconsistencies to the developers who indeed resolved 5 of them.

Zhong *et al.* (2011) automatically generate specifications from API documentation concerning resource usage, namely creation, lock, manipulation, unlock, and closure. The authors then identify source code not complying with the generated specification and report it as defect. An example of defect is manipulating a file resource but never closing it. The source code analysis is performed by building control flow graphs. They contacted developers of the open-source projects who confirmed 5 previously unknown defects.

Pradel et Gross (2013) propose a technique to detect anomalies in equally typed method arguments. The authors search for inconsistencies between the formal and actual method parameters. In particular, they extract identifiers from the method definition and the different call sites. An anomaly in a call site occurs when the order suggested by the names of the actual parameters is inconsistent with the order suggested by the names of the formal parameters of the method. To reduce the number of false positives, an anomaly is only reported if reordering the actual parameters fits better the order used in other call sites. A naming bug is detected if the different call sites follow a naming scheme but not the method definition. A prototype implementation for Java and C/C++ projects is provided as an Eclipse plugin.

Discussion

We share with Lawrie *et al.* (2006c) the conviction that the consistency between code and comments is an important quality indicator. While they measure consistency in terms of cosine similarity, the LAs defined in Chapter 6 of this dissertation focus on identifying practices that break the consistency—e.g., *Not implemented condition* (B.1) and *Method signature and comment are opposite* (C.2).

While the approaches proposed by Tan *et al.* (2007, 2011, 2012); Zhong *et al.* (2011)

address inconsistencies specific to certain source code aspect/implementation technology—i.e., lock/call, null values/exceptions, synchronization, and resource usage—LAs can be considered as complementary as they deal with generic naming and commenting issues that can arise in OO code, specifically in methods and attributes.

We have in common some of the inconsistencies with the naming bugs defined by Høst et Østvold (2009). In particular, we share *"Set" method returns* (A.3) and the general idea behind *"Get" method does not return* (B.3) and *Not answered question* (B.4). However, the main difference is that LAs defined here also consider attributes, identify inconsistencies between comments and signatures, and identify opposite meaning inconsistencies. In addition, our prototype detection tool, LAPD, considers the comment before reporting an inconsistency to check whether the particular inconsistency is documented. For example, if the comment of a method containing the LA *"Set" method returns* (A.3) documents that the returned value is for example the old value of the attribute then LAPD will not report the method to the developer as the unusual behavior is documented and thus unlikely to cause confusion. Finally, LAPD analyses source code whereas Lancelot works on bytecode thus can only analyze code that compiles.

Finally, our contribution with respect to previous work is that we evaluate the relevancy of the defined LAs by conducting two studies with developers—*external* and *internal* developers—and show that the majority of them consider LAs as poor practices.

3.4 Developers' Perceptions of Code Smells

Mäntylä et Lassenius (2006) conducted an empirical study on the subjective evaluation of code smells that identify poorly evolvable structures in software. They asked industrial software developers of a Finnish company to evaluate how much of each code smell existed in a particular software module they are familiar with. The human evaluations were then compared to the detected smells using code metrics. Overall, they noticed that demographic data partially explains the variance of human evaluations. For example, they found that lead developers saw more structured smells as opposed to the other developers who saw more code smells.

Yamashita et Moonen (2013) performed a study—involving 85 professionals—with the aim of investigating the perception of code smells, in particular, the degree of awareness of code smells, their severity, and the usefulness of automatic tool support. Surprisingly, 23 of the participants (32%) were not aware of such code smells. From the remaining 50 participants, i.e., those that have at least heard of antipatterns and code smells, only 3 participants (6%) were not concerned about the presence of code smells. 47 of the participants (94%) were

concerned at a different level—10 (20%) were slightly concerned, 11 (22%) were somewhat concerned, 19 (38%) were moderately concerned, and 7 (14%) were extremely concerned. Yamashita and Moonen performed categorical regression analysis and found that the more familiar participants are with antipatterns and code smells, the more concerned they are.

Palomba *et al.* (2014) also studied developers' perceptions of code smells. They evaluated examples of 12 code smells found in 3 open-source Java projects from the perspective of 34 external and internal developers. Their results show that there are some code smells that developers do not perceive as poor practices. They also observed that for several code smells experienced developers are more concerned than less experienced developers.

Discussion

We share with the previous works the interest in how developers perceive poor practices. The main difference between previous works and our work is that while they evaluate practices that have been out there for more than a decade, we study practices that developers were not at all familiar with—i.e., study with external developers (see Chapter 7)—or just introduced to—i.e., study with internal developers (see Chapter 8). This could be one of the reasons why, compared to the results of Yamashita et Moonen (2013), a lower number of participants perceive LAs as 'Poor' or 'Very Poor'—69% and 51% for external and internal developers respectively—as opposed to antipatterns and code smells—94% when only considering participants familiar with antipatterns and code smells.

CHAPTER 4

IDENTIFIER TERM DISPERSION AND CODE QUALITY

Highlight: Our hypothesis is that the poor quality of the source code lexicon contributes to the faultiness of a program entity. We define a measure, named High Entropy and High Context Coverage (HEHCC), characterizing the physical and conceptual dispersions of terms composing source code identifiers. We investigate whether program entities that contain HEHCC terms—i.e., terms used in many places and in different contexts—are more fault prone. As it is well known that the size of an entity is one of the best fault predictors, we also evaluate whether the quality of the lexicon—as measured by terms dispersion—help structural metrics—such as LOC—to explain software faults.

The identification of faulty source code entities is generally based on product metrics, such as size, cohesion, or coupling (Gyimóthy *et al.*, 2005; Liu *et al.*, 2009; Marcus *et al.*, 2008) as well as process-oriented metrics, such as number of file changes (Zimmermann *et al.*, 2007). However, we believe that fault proneness is a complex phenomenon hardly captured solely by structural characteristics of code entities. Indeed, several studies showed that identifiers impact program comprehension (Takang *et al.*, 1996; Deißenböck et Pizka, 2006; Haiduc et Marcus, 2008; Binkley *et al.*, 2009) and code quality (Marcus *et al.*, 2008; Poshyvanyk et Marcus, 2006; Butler *et al.*, 2009). We concur with Deißenböck and Pizka's observation that proper identifiers improve quality and that identifiers should be used consistently (Deißenböck et Pizka, 2006). Source code with high quality identifiers, carefully chosen and consistently used in their contexts, likely ease program comprehension and support developers in building consistent and coherent conceptual models.

In this chapter, we present, to the best of our knowledge, the first empirical study on the relation between the terms composing identifiers and fault proneness. Terms are identifier components (e.g., `get` and `String` are the terms composing `getString`). We conjecture that a term should carry a single meaning in the context where it is used. Terms referring to different concepts or used inconsistently in different contexts may increase the program comprehension burden by creating a mismatch between the developers' cognitive model and the intended meaning of the term, thus ultimately increasing the risk of fault proneness.

Context definition is left intentionally blurred as it may stand for a code region (e.g., method or attribute, class or component) as well as for the developers' knowledge and mental models

of any given code region or artifact. Misunderstanding impacts the cognitive process and is difficult to quantify. We use linguistic information extracted from a given code region as a surrogate of the developers' mental models. More precisely, linguistic information extracted from methods and attributes is used to quantify *term entropy* and *context coverage*. Term entropy is derived from entropy in information theory and measures the "physical" dispersion of a term in a program, i.e., the higher the entropy, the more scattered the term is across entities. Term context coverage exploits IR (Baeza-Yates et Ribeiro-Neto, 1999) techniques to measures the "conceptual" dispersion of the entities in which the term appears. The higher the context coverage of a term, the more unrelated are the linguistic information and, possibly, the concepts of the corresponding entities.

To support our conjecture that terms with high entropy and high context coverage may help to locate fault-prone methods and attributes we report a preliminary case study on two open-source projects: *ArgoUML* and *Rhino*. We show that there is a statistically significant relation between *term entropy*, *context coverage*, and odds ratio that an entity is fault prone. Thus, the contributions of this chapter can be summarized as follows:

— a novel measure characterizing the "physical" (entropy) and "conceptual" (context) dispersions of terms;

— a preliminary empirical study showing the relation between entropy and context coverage with fault proneness.

4.1 Measure the Physical and Conceptual Dispersion of Terms

To calculate term entropy and context coverage we extract identifiers, split them into terms, and build a *term-by-entity* matrix. The generic entry $a_{i,j}$ of the term-by-entity matrix denotes the number of occurrences of the i^{th} term in the j^{th} entity.

4.1.1 Term Entropy

Entropy of a discrete random variable measures the amount of uncertainty (Cover et Thomas, 2006). To compute the term entropy, we consider terms as random variables with some associated probability distributions. Given a term, its entries in the term-by-entity matrix are the counts of term occurrences and thus by normalizing over the sum of its row entries a probability distribution for each term is obtained. A normalized entry $\widehat{a}_{i,j}$ is then the probability of the presence of the term t_i in the j^{th} entity. We then compute term entropy as:

$$H(t_i) = -\sum_{j=1}^{n}(\widehat{a}_{i,j}) \cdot log(\widehat{a}_{i,j}) \quad i = 1, 2, \ldots, m$$

With term entropy, the more scattered among entities a term is, the closer to the uniform distribution is its mass probability and, thus, the higher is its entropy. On the contrary, if a term has a high probability to appear in few entities, then its entropy value will be low.

4.1.2 Term Context Coverage

The context coverage of term t_k (where $k = 1, 2, \ldots, m$) is computed as the average textual similarity of the contexts (here entities) containing t_k:

$$CC(t_k) = 1 - \frac{1}{N} \sum_{e_i, e_j \in C} sim(e_i, e_j)$$

where $C = \{e_l | a_{k,p} \neq 0\}$ is the set of all entities in which term t_k occurs, N is the number of compared entities, and $sim(e_i, e_j)$ represents the textual similarity between entities e_i and e_j computed using Latent Semantic Indexing (LSI) (Deerwester *et al.*, 1990), an advanced IR method, with a subspace dimension fixed to 100. A low value of the context coverage of a term means a high similarity between the entities in which the term appears, i.e., the term is used in consistent contexts.

4.1.3 Aggregate Metric

In this preliminary investigation, we use the variable *numHEHCC*, number of high entropy and high context coverage, associated with all entities and defined as:

$$numHEHCC(E_j) = \sum_{i=1}^{m} a_{ij} \cdot \psi(H(t_i) \geq th_H \wedge CC(t_i) \geq th_{CC})$$

where a_{ij} is the frequency in the term-by-entity matrix of term t_i in entity E_j ($j = 1, 2, \ldots, n$) and $\psi()$ is a function returning one if the passed Boolean value is true; zero otherwise. Thus, $numHEHCC$ represents the overall number of times any term with high entropy (value above th_H) and high context coverage (value above th_{CC}) is found inside an entity. This aggregate metric is used throughout the following case study to compute correlation, build linear, logistic models, and contingency tables.

4.2 Case Study Design

It is well known that the size of an entity is one of the best fault predictors (Gyimóthy *et al.*, 2005). Thus, we first verify that $numHEHCC$ is somehow at least partially complementary to size. Second, we believe that it is important to understand if entropy and context

coverage help to locate entities likely to be fault prone when changed. Therefore, the case study is designed to answer the following research questions:

RQ1: *Do Term Entropy and Context Coverage Capture Characteristics Different from Size?*

RQ2: *Do Term Entropy and Context Coverage Help to Explain the Presence of Faults in an Entity?*

In **RQ1**, our goal is only to validate that the newly proposed metric brings information complementary to the information captured by LOC. However, the real focus of this work is to answer **RQ2**: evaluating whether the proposed metric, alone, can be used to explain fault proneness. To answer **RQ2**, we assess a risk in terms of odds ratio, and thus, do not claim any causation.

4.2.1 Objects

The context of the study are two open-source projects: *Rhino* and *ArgoUML*. The choice of this two projects is twofold: they were previously used in other case studies, e.g., (Eaddy *et al.*, 2008), and a mapping between faults and entities (attributes and methods) is available (Eaddy *et al.*, 2008; Thummalapenta *et al.*, 2010). We select the version of *ArgoUML* that has the maximum number of faulty entities (v0.16) and one of the versions of *Rhino* with low number of bugs (v1.4R3). Details regarding the projects can be found in Appendix B.

4.2.2 Data Collection

To calculate term entropy and context coverage, we extract identifiers found in class attributes and methods (e.g., names of variables, methods, method parameters). We split identifiers using a CamelCase splitter to build the term dictionary (e.g., `getText` is split into `get` and `text`) and we retain terms whose length is at least two characters.

We reused the mapping between faults and entities (attributes and methods) from previous studies (Eaddy *et al.*, 2008; Thummalapenta *et al.*, 2010).

4.2.3 Analysis Method

RQ1: Do Term Entropy and Context Coverage Capture Characteristics Different from Size?

To statistically analyze **RQ1**, we compute the correlation between the size measured in LOC and $numHEHCC$. Then, we estimate the linear regression models between LOC (independent variable) and $numHEHCC$ (dependent variable). Finally, as an alternative to the Analysis Of Variance (ANOVA) (Sheskin, 2007) for dichotomous variables, we build

logistic regression models between fault proneness (explained variable) and LOC and the proposed new metric (explanatory variables). Thus, we formulate the null hypothesis:

H_{0_1}: numHEHCC does not capture a dimension different from LOC.

We expect that some correlation with size does exist: longer entities may contain more terms with more chance to have high entropy and high context coverage.

RQ2: Do Term Entropy and Context Coverage Help to Explain the Presence of Faults in an Entity?

For **RQ2**, we formulate the following null hypothesis:

H_{0_2}: There is no relation between numHEHCC and fault proneness.

We use a Prop Test (Sheskin, 2007) to test the null hypothesis. If *numHEHCC* is important to explain fault proneness, then the Prop Test should reject the null hypothesis with a statistically significant p-value.

To quantify the effect size of the difference between entities with and without high values of term entropy and context coverage, we also compute the *odds ratio* (OR) (Sheskin, 2007) indicating the likelihood of faulty entities to contain terms with high entropy and context coverage.

The term context coverage distribution is skewed toward high values. For this reason, we use the 10% highest values of term context coverage to define a threshold identifying the high context coverage property. We do not observe a similar skew for the values of term entropy and, thus, the threshold for high entropy values is based on the standard outlier definition—1.5 times the inter-quartile range above the 75% percentile (Fenton et Pfleeger, 1996). The thresholds values chosen for entropy and context coverage in our study are 4.17 and 0.79 for *Rhino*, and 4.9, and 0.83 for *ArgoUML*, respectively.

4.3 Results

We now discuss the results aiming at providing answers to our research questions.

4.3.1 RQ1: Do Term Entropy and Context Coverage Capture Characteristics Different from Size?

Table 4.1 reports the results of Spearman's correlation for both projects. As expected, some correlation exists between LOC and *numHEHCC*. Despite a 40% correlation a linear regression model built between *numHEHCC* (dependent variable) and LOC (independent variable) attains an R^2 lower than 19% (see Table 4.2). The R^2 coefficient can be interpreted

Table 4.1 Correlation tests.

Project	Correlation	p-values
ArgoUML	0.3646527	$\prec 2.2e-16$
Rhino	0.4467815	$\prec 2.2e-16$

as the percentage of variance of the data explained by the model and thus $1 - R^2$ is an approximation of the unexplained variance of the model. Thus, Table 4.2 supports the conjecture that LOC does not substantially explain $numHEHCC$. Correlation and linear regression models can be considered as further verification that LOC and $numHEHCC$ help to explain different dimensions of fault proneness.

The relevance of $numHEHCC$ in explaining faults is further supported by logistic regression models. Table 4.3 reports the interaction model built between fault proneness (dependent variable) and the explanatory variables LOC and $numHEHCC$. In both models, $M_{ArgoUML}$ and M_{Rhino}, the intercept is relevant as well as $numHEHCC$. Most noticeably in *Rhino* the LOC coefficient is not statistically significant as well as the interaction term, $LOC : numHEHCC$. This lack of significance is limited to *Rhino*: for *ArgoUML*, both LOC and the interaction term are statistically significant. In both models, $M_{ArgoUML}$ and M_{Rhino}, the LOC coefficient is, at least, one order of magnitude smaller than the $numHEHCC$ coefficient. This difference can partially be explained by the different range of LOC versus $numHEHCC$. On average, in both projects, method size is below 100 LOC and most often a method contains one or two terms with high entropy and context coverage. Thus, conservatively, we can say that both LOC and $numHEHCC$ have the same impact in terms of probability. In other words, the models in Table 4.3 support the conjecture that $numHEHCC$ helps to explain fault proneness. Based on the reported results, we can conclude that although some correlation exists between LOC and $numHEHCC$, statistical evidence allows us to reject, on both projects, the null hypothesis H_{0_1}.

4.3.2 RQ2: Do Term Entropy and Context Coverage Help to Explain the Presence of Faults in an Entity?

To answer **RQ2**, we perform prop-tests and test the null hypothesis H_{0_2}. Indeed, (i) if prop-tests reveal that $numHEHCC$ divides the population into two sub-populations and (ii) if the sub-population with positive values for $numHEHCC$ has an odds ratio bigger than one, then $numHEHCC$ may act as a risk indicator. For entities with positive $numHEHCC$, it will be possible to identify those terms leading to high entropy and high context coverage, identifying also their contexts and performing refactoring actions to reduce entropy and high context coverage.

Table 4.2 Linear regression models.

	Variables	Coefficients	p-values
Rhino ($R^2 = 0.1891$)	Intercept	0.038647	0.439
	LOC	0.022976	$\prec 2e-16$
ArgoUML (R^2=0.1665)	Intercept	-0.0432638	0.0153
	LOC	0.0452895	$\prec 2e-16$

Table 4.3 Logistic regression models.

	Variables	Coefficients	p-values
$M_{ArgoUML}$	Intercept	-1.688e+00	$\prec 2e-16$
	LOC	7.703e-03	$8.34e-10$
	numHEHCC	7.490e-02	$1.42e-05$
	LOC:numHEHCC	-2.819e-04	0.000211
M_{Rhino}	Intercept	-4.9625130	$\prec 2e-16$
	LOC	0.0041486	0.17100
	numHEHCC	0.2446853	0.00310
	LOC:numHEHCC	-0.0004976	0.29788

Tables 4.4 and 4.5 show the confusion matrices for the two projects, together with the corresponding p-value and odds ratios. The number of fault prone entities for *Rhino* being very low, we also compute the Fisher exact test, which is commonly used as an alternative for small samples. As the tables show, the null hypothesis H_{0_2} can be rejected.

We further investigate the odds ratio of entities containing two or more terms with high entropy and high context coverage with those entities that only contain one such term. The results are not statistically significant (with an OR close to one). These results suggest that the difference between fault-prone entities and others is between not containing high entropy and high context coverage terms and containing one or more such terms.

We also analyze the odds change for LOC and $numHEHCC$. For example, in the case of *ArgoUML* for a fixed LOC, one unit increase of $numHEHCC$ has almost the same odds effect than an increase of 10 LOCs. In the case of *Rhino*, for a fixed size of entities, one unit increase of $numHEHCC$ has more effect than an increase of 50 LOCs.

4.4 Discussions

The results support the conjecture that term entropy and context coverage only partially correlate with size. We also show that the number of high entropy and high context coverage

Table 4.4 *ArgoUML* v0.16 confusion matrix.

ArgoUML	numHEHCC ≥ 1	numHEHCC $= 0$	Total
Fault prone	381	1706	2087
Fault free	977	9359	10336
Total	1358	11065	12423
Prop-test: p-value $\prec 2.2e-16$; Odds ratio $= 2.139345$			

Table 4.5 *Rhino* v1.4R3 confusion matrix.

Rhino	numHEHCC ≥ 1	numHEHCC $= 0$	Total
Fault prone	6	8	14
Fault free	172	1438	1610
Total	178	1446	1624
Prop-test: p-value $= 0.0006561$; Odds ratio $= 6.270349$			
Fisher's Exact Test: p-value $= 0.002258$			

terms contained in a method or attribute helps to explain the probability of it being faulty. Furthermore, the odds ratio of being faulty for a method (or attribute) containing one or more terms with high entropy and high context coverage is six and two for *Rhino* and *ArgoUML*, respectively: if a *Rhino* method contains an identifier with a term having high entropy and high context its probability of being faulty is six times higher; for *ArgoUML* the probability of being faulty is two times higher.

4.4.1 Threats to Validity

Conclusion validity: Proper non-parametric statistical tests are used and the null hypothesis is rejected with a significant p-value. We perform an additional test—Fisher exact test—for *Rhino* as the number of faulty entities is small.

Internal validity: We use manually validated faults that have been used in previous studies but we cannot claim that all fault prone entities have been correctly tagged or that no fault prone entity has been missed. We used a threshold to identify terms with high entropy and context coverage and compute $numHEHCC$, which could influence the results.

Construct validity: To compute term context coverage we use the textual similarity between entities using LSI. Although LSI is known to deal with synonymy and polysemy, a domain ontology such as WordNet (Miller, 1995) may lead to a more accurate context representation.

External validity: The study is limited to two projects, *ArgoUML* 0.16 and *Rhino* 1.4R3. Results are encouraging but replications are needed to increase the generalizability of the results achieved.

Reliability validity: We use open-source projects whose source code is available. We attempt to provide all necessary details to replicate the analysis.

4.5 Conclusion

In this chapter we introduced *term entropy* and *context coverage* to measure how scattered terms are across program entities and how unrelated are the methods and attributes containing these terms. We provided mathematical definitions of term entropy and context coverage and reported a preliminary case study involving two open-source projects: *ArgoUML* and *Rhino*. We show that the newly proposed lexicon metric, i.e., *numHEHCC*, capture additional information not captured by LOC, i.e., a widely used metric for the size of an entity. We also show that this additional information is an asset for fault explanation by showing that the probability of an entity containing high entropy and high context coverage terms to contain a fault is higher than the probability of an entity without such terms.

CHAPTER 5

LEXICON BAD SMELLS (LBS) AND CODE QUALITY

Highlight: In Chapter 4, we provided evidence the quality of the lexicon—as measured by terms dispersion—helps structural metrics—such as LOC—to explain software faults. In this chapter, we use different lexicon and structural measures and take the analysis a step further—i.e., from fault explanation to fault prediction. Thus our objective is to evaluate whether the quality of the lexicon—as measured by LBS—help structural metrics—such as CK—to predict faulty classes.

The cost of identifying and fixing faults in a project already in production may be extremely high. To avoid such costs, developers spend a large portion of the project development time on testing, to identify faulty classes prior to release. To assist developers in this respect, various studies have been conducted in the research community measuring the quality of the source code using structural metrics (Basili *et al.*, 1996; Zhou et Leung, 2006; Zimmermann *et al.*, 2007; Mende et Koschke, 2009), process metrics (Nagappan et Ball, 2005; Hassan, 2009) or previous faults (Kim *et al.*, 2007; Weyuker *et al.*, 2010). Structural metrics are a lightweight alternative and they have been shown to have good performance for fault prediction (D'Ambros *et al.*, 2010).

The Chidamber and Kemerer object-oriented metrics suite (Chidamber et Kemerer, 1994) is widely used as a representative of structural metrics. The underlying idea for using these metrics is that if the code is complex, it will be also difficult to understand and maintain; hence, it is susceptible to the introduction of faults. The CK metrics are based on information about the *structure* of the source code. Besides the structural complexity, other researchers have shown the importance of source code identifiers Deißenböck et Pizka (2006); Haiduc et Marcus (2008); Butler *et al.* (2009). We concur with those works and believe that the lexicon used in naming identifiers has an impact on the understandability of the code. To measure the linguistic quality of identifiers we use the catalog of Lexicon Bad Smells (LBS) defined by Abebe *et al.* (2009b). LBS are potential identifier construction problems that can compromise the quality of the identifier and hence hinder program comprehension. We conjecture that adding such information to the structural metrics used in fault proneness prediction will improve the prediction. In this study, we investigate if this conjecture holds or not. Prior to such investigation, as a sanity check we have assessed whether LBS add any new information with respect to the CK metrics; results are positive. To conduct the prediction, we first

identify the best model that can be obtained with the CK metrics and then we investigate whether adding LBS to the CK metrics improves the prediction. The results indicate that there is an improvement in the majority of the cases. Following these results, we also carry a study to identify those LBS that contribute the most to the improvement of the prediction.

5.1 Case Study Design

Structural metrics measure different aspects of the code that can be used to predict fault proneness of a class. In this study, we conjecture that the quality of identifiers has also an impact on the fault proneness of a class, besides the structural metrics. To prove this conjecture, we have formulated the following three research questions:

RQ1: *Do LBS Bring New Information with Respect to Structural Metrics?*

RQ2: *Do LBS Improve Fault Prediction?*

RQ3: *Which LBS Help More to Explain Faults?*

In **RQ1**, our goal is only to validate that the LBS bring information complementary to the information captured by structural metrics. However, the real focus of this work is to answer **RQ2**: evaluating whether LBS can improve fault prediction when used in addition to structural metrics. Finally, as several LBS exist, in **RQ3** we are interested to understand which LBS are more helpful to predict faults and thus must be avoided.

5.1.1 Objects

For our case study, we have considered three open-source projects written in Java, namely *ArgoUML*, *Eclipse*, and *Rhino*. As in Chapter 4, the choice of the projects was mainly driven by the availability of fault prone data (Khomh *et al.*, 2012). Details regarding the projects can be found in Appendix B.

5.1.2 Analysis Method

In the following we describe the experimental setting of our study, starting with the dependent and independent variables and then continuing with the setting for each research question.

Variables

For building the prediction models we considered the following variables: As *dependent* variable we use HASB, a dichotomous variable indicating whether a class is faulty or not.

The overall set of *independent* variables consists of a set of structural metrics as considered by Kpodjedo *et al.* (2011) (see Table 5.1) and the LBS defined by Abebe *et al.* (2009b) (see Table 5.2).

Table 5.1 List of considered structural metrics.

Acronym	**Description**
CBO	Coupling between objects
DIT	Depth of Inheritance Tree
LCOM1	Lack of COhesion in Methods 1
LCOM2	Lack of COhesion in Methods 2
LCOM5	Lack of COhesion in Methods 5
LOC	Line Of Code
NAD	Number of Attributes Declared
NMD	Number of Methods Declared
NOC	Number Of Children
RFC	Response For a Class
WMC	Weighted Methods per Class

Table 5.2 List of considered lexicon metrics.

Description
Extreme contraction
Inconsistent identifier use
Meaningless terms
Misspelling
Odd grammatical structure
Overloaded identifiers
Useless type indication
Whole-part
Synonyms and similar identifiers
Terms in wrong context
No hyponymy/hypernymy in class hierarchies
Identifier construction rules

RQ1: Do LBS Bring New Information with Respect to Structural Metrics?

In the first research question, **RQ1**, we investigate if LBS measure the same aspects of the code as structural metrics or not. To carry out this investigation, following Marcus *et al.* (2008), we use PCA. PCA aggregates the metrics into few orthogonal components called principal components (PC). We use the information captured in the PCs to analyze and answer **RQ1**. In particular, we analyze the following two aspects of the PCs: i) the number of case study versions in which an LBS contributes to at least one retained PC, and ii) the

number of case study versions in which an LBS is the major contributor to at least one retained PC.

To select a subset of the PC we used a threshold of 95%—similar to Marcus *et al.* (2008). That is, we retained the components that explain up to 95% of the variance. For each principal component, we rank the attributes based of their importance (weight) and we apply a 10% relative threshold to decide which attributes contribute to the component. In other words, if the importance of attribute a_j drops with more than 10% of the importance of the preceding attribute a_i, then a_j and all other following attributes will be discarded for the particular component. If LBS bring new information with respect to structural metrics then LBS will be kept in the retained principal components and will give major contributions to them. To answer this research question we analyze two aspects: i) the number of case study versions in which an LBS contributes to at least one retained PC, and ii) the number of case study versions in which an LBS is the major contributor of at least one retained PC.

RQ2: Do LBS Improve Fault Prediction?

In **RQ2**, we investigate if our conjecture holds by assessing the contribution of LBS, in addition to the structural metrics, in improving the prediction capability of a model. To assess the contribution of LBS, we carry out predictions using as independent variables, on the one hand, only structural metrics, and on the other hand, structural metrics plus LBS. We consider three models, namely Logistic Regression Model (LRM), Random Forest (RF), and Support Vector Machine (SVM). The capability of prediction is then evaluated using the performance measures described in Section 2.4.3. We then compare the results using the achieved net improvements and the average delta percentage. Prior to the comparison of the two sets of independent variables, we compare and select the best model in predicting fault prone classes using only the CK metrics.

Settings When building a LRM we perform backward variable elimination and predict using the retained variables. In addition, before prediction we control for multicollinearity by removing variables with a VIF greater than 2.5 (Shihab *et al.*, 2010; Cataldo *et al.*, 2009). The following settings are common for all models: As Gyimóthy *et al.* (2005) we standardize all metrics before performing the calculations (i.e., zero mean and unit variance). Like Kamei *et al.* (2010), for each type of model, we predict faulty classes in two configurations: within the same version and for the next version. Prediction within the same version represents scenarios in which there is no prior record of buggy classes while the latter represents scenarios in which such data is available. When predicting within the same version, we use 10-fold cross validation. For each configuration we build two models: one where the independent

variables are the CK set of metrics alone and the second where the independent variables are CK and LBS.

RQ3: Which LBS Help More to Explain Faults?

The last research question, **RQ3**, focuses on identifying those LBS that contribute the most to the prediction of fault prone classes. To answer this research question, we use the weights assigned to each LBS by the model and we compute the median rank of each LBS.

To decide which LBS best help for fault prediction we rank the attributes based on their importance in the best model selected in **RQ2**. We then calculate the median rank across the versions of the project and select the top three LBS separately for each project.

5.1.3 Data Collection

We reuse the data regarding the fault proneness of classes (i.e., dependent variable) that have been previously published by Khomh *et al.* (2012). To collect the data for the independent variables we proceed as follows. We calculate the set of CK metrics using the Primitives Operators Metrics (POM) framework (Guéhéneuc *et al.*, 2004). To identify LBS, we have used a suite of tools called *LBSDetectors* [1] which have been developed for use in previous studies (Abebe *et al.*, 2009b, 2011).

5.2 Results

5.2.1 RQ1: Do LBS Bring New Information with Respect to Structural Metrics?

To answer this research question, we summarize the percentages of case study versions in which an LBS contributes to at least one retained PC (Table 5.3) and the percentages of case study versions in which an LBS is the major contributor to at least one retained PC, i.e., the LBS was ranked first (Table 5.4). Table 5.5 shows in details the result of PCA for *ArgoUML* v0.16. In particular, we show the weight and ranking (in parentheses) of the attributes after the relative threshold is applied.

ArgoUML For all versions of *ArgoUML* we retained between 11 and 13 principal components that explain at least 95% of the variance. Two LBS attributes were kept in at least one PC in all versions and those are: *inconsistent terms* and *useless types*. Between them, *useless types* was the major contributor of at least one PC in all versions.

1. http://selab.fbk.eu/LexiconBadSmellWiki/

Rhino The number of components that explain at least 95% of the variance for *Rhino* is the same as for *ArgoUML*. Five LBS attributes were kept in at least one PC in all versions and those are: *inconsistent terms*, *synonym similar*, *odd grammatical structure*, *overloaded identifiers*, and *meaningless*. As in *ArgoUML*, one LBS attribute was present as a major contributor in all versions and this is *overloaded identifiers*.

Eclipse The number of retained PC is between 13 and 14. The six LBS that are present in all versions are: *inconsistent terms*, *odd grammatical structure*, *extreme contraction*, *overloaded identifiers*, *useless types*, and *meaningless*. The majority of them (four) are ranked first: *inconsistent terms*, *extreme contraction*, *overloaded identifiers*, and *meaningless*.

Overall All LBS were present in more than 50% of the analyzed projects. *Inconsistent terms* was present in at least one dimension in all analyzed versions meaning that it is the major LBS attribute that helps to explain a new variability dimension. Another different variability dimension in most cases seems to be captured by *overloaded identifiers* and *useless types*.

5.2.2 RQ2: Do LBS Improve Fault Prediction?

When dealing with multiple independent variables, we must account for possible correlations among them. We compute the Spearman's correlation coefficient (r_s) between all possible pairs of metrics for all metrics. In Table 5.6 we summarize the number of projects for which each pair of metric has a statistically significant strong correlation—i.e., p-value ≤ 0.05 and $r_s \geq 0.8$ or $r_s \leq -0.8$—where the thresholds are based on previous studies (Al Dallal et Briand, 2012)). The table shows only metrics with positive values, i.e., metrics for which a statistically significant strong correlation is observed in at least one project. Results are consistent with the results of RQ1 as the strongly correlated metrics are grouped into one component (PC1 in Table 5.5).

For each evaluation metric, Table 5.7 shows the average values scored by the corresponding model for both configurations for prediction (same and next version). CK metrics are used to build the prediction models. The values in bold are the best values of the three models

Table 5.3 LBS retained in the principal components.

Project	*Misspelling*	*Inconsistent terms*	*Synonym similar*	*Odd grammatical structure*	*Extreme contraction*	*Overloaded identifiers*	*Identifier construction*	*Useless types*	*Meaningless terms*
Eclipse	0.0%	100.0%	40.0%	100.0%	100.0%	100.0%	80.0%	100.0%	100.0%
ArgoUML	66.7%	100.0%	50.0%	66.7%	66.7%	83.3%	83.3%	100.0%	16.7%
Rhino	87.5%	100.0%	100.0%	100.0%	75.0%	100.0%	62.5%	87.5%	100.0%
All	57.9%	100.0%	68.4%	89.5%	78.9%	94.7%	73.7%	94.7%	73.7%

Table 5.4 LBS ranked first in the retained principal components.

Project	*Misspelling*	*Inconsistent terms*	*Synonym similar*	*Odd grammatical structure*	*Extreme contraction*	*Overloaded identifiers*	*Identifier construction*	*Useless types*	*Meaningless terms*
Eclipse	0.0%	100.0%	20.0%	20.0%	100.0%	100.0%	80.0%	80.0%	100.0%
ArgoUML	50.0%	83.3%	0.0%	16.7%	33.3%	83.3%	66.7%	100.0%	16.7%
Rhino	0.0%	87.5%	50.0%	0.0%	50.0%	100.0%	62.5%	62.5%	87.5%
Overall	15.8%	89.5%	26.3%	10.5%	57.9%	94.7%	68.4%	78.9%	68.4%

Table 5.5 Detailed results of PCA for *ArgoUML* v0.16.

PC	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
Cumulative proportion	40.9%	51.8%	59.88%	65.54%	71.06%	76.2%	81.02%	85.29%	89.33%	92.91%	95.53%
CBO	**0.275(9)**	0.203	0.35	0.0741	0.0176	0.0994	0.0954	0.11	0.0853	0.0607	0.0953
DIT	0.0311	0.0551	0.123	0.13	**0.772(1)**	0.46	0.0998	0.338	0.0457	0.0665	0.107
LCOM1	**0.281(7)**	0.36	0.0835	0.0277	0.00268	0.0387	0.0503	0.0812	0.184	0.28	0.0641
LCOM2	**0.278(8)**	0.366	0.0879	0.0272	0.00323	0.0382	0.0507	0.0807	0.188	0.282	0.0685
LCOM5	0.111	0.307	0.00385	0.0976	0.206	0.269	0.0478	**0.753(1)**	0.35	0.16	0.0615
LOC	**0.29(5)**	0.15	0.367	0.0276	0.0123	0.0217	0.0729	0.165	0.0668	0.0341	0.134
NAD	0.21	0.101	**0.442(1)**	0.0404	0.0119	0.0138	0.0763	0.0568	0.368	0.0434	**0.604(1)**
NMD	**0.338(1)**	0.0988	0.0846	0.0403	0.0618	0.00984	0.0373	0.0147	0.0764	0.0998	0.108
NOC	0.0205	0.107	0.0854	0.386	0.428	**0.774(1)**	0.00912	0.113	0.132	0.0118	0.0585
RFC	**0.296(4)**	0.176	0.274	0.0458	0.0453	0.00323	0.0397	0.0971	0.0342	0.0189	0.0197
WMC	**0.318(2)**	0.12	0.286	0.0338	0.0116	0.0181	0.0958	0.0996	0.0384	0.0614	0.131
misspelling	0.24	0.201	0.187	0.207	0.255	0.0979	0.0973	0.0529	0.0174	0.211	**0.571(2)**
inconsistent terms	0.205	0.246	0.147	0.0116	0.0484	0.0543	0.383	0.29	0.189	**0.6(1)**	0.178
synonym similar	**0.288(6)**	0.314	0.102	0.00884	0.00461	0.000561	0.155	0.0958	0.0317	0.154	0.241
odd grammatical structure	**0.305(3)**	0.0148	0.28	0.013	0.048	0.0274	0.173	0.0301	0.0203	0.0892	0.152
extreme contraction	0.0772	0.253	0.266	**0.592(1)**	0.276	0.212	0.0624	0.166	0.336	0.288	0.15
overloaded identifiers	0.144	0.0224	0.00102	0.236	0.161	0.00659	**0.802(1)**	0.0575	0.00447	0.467	0.0241
identifier construction	0.14	**0.416(1)**	0.271	0.266	0.0227	0.143	0.0104	0.0399	0.41	0.139	0.299
useless types	0.0413	0.248	0.236	**0.539(2)**	0.0042	0.153	0.304	0.318	**0.561(1)**	0.186	0.00662

Table 5.6 Number of projects with statistically significant strong correlation.

	CBO	LCOM1	LCOM2	LOC	NMD	RFC
LCOM2		13				
LOC	3	13				
NMD		19	5	8		
RFC	3	4		14	3	
WMC	3	13		19	15	14

considered for the given metric. For all projects, SVM scores first for the majority of the evaluation metrics. Hence, we have based our investigation on the contribution of LBS to the improvement of fault prediction using SVM.

Table 5.8 shows the number of versions in which CK plus LBS metrics improve, decrease or keep the prediction unchanged, when compared to CK metrics alone. The last two columns show the net improvement within/across versions and the average delta percentage of LBS plus CK metrics over CK alone for the various evaluation metrics. Positive values of net improvements, for all types of evaluation metrics, indicate that in the majority of the versions CK plus LBS are better predictors than CK alone, while negative values indicate the opposite. A zero net improvement means that both sets of independent variables were found better than the other in an equal number of versions or that they are equal in all versions. For all evaluation metrics except *absolute error*, the same is true for the average delta percentage, which is computed on the average values over all versions of the corresponding project. For *absolute error*, a negative value means that there is a reduction in the amount of error and hence indicates an improvement while the opposite holds for positive values of *absolute error*.

The predictions using CK plus LBS metrics have outperformed those of CK alone in most of the versions of the three projects, when considering both within and across version prediction. For *ArgoUML*, the prediction on the same versions using CK and LBS together has improved in at least 5 of the 6 versions considered, according to the different evaluation metrics. For *Eclipse* the improvement observed in all versions is consistently reported by all evaluation metrics. Figure 5.1 shows the average values of all versions of *Eclipse* for the evaluation metrics. We observe an important improvement for all metrics except for *accuracy* where the improvement is minor. The result of the evaluation metrics for *Rhino* shows that there is an improvement in the majority of the versions considered (at least 6 out of 8). The distributions of the evaluation metrics for all projects are shown in Figure 5.3.

When predicting on the next version, results depend on the evaluation metrics. For *ArgoUML*, negative net improvement values are observed in three of the evaluation metrics while the other three show that there is a net improvement in at least 3 out of the 5 versions predicted. For *Eclipse* we observe a negative net improvement only for one of the evaluation metrics and a positive net improvement for three of the evaluation metrics. Figure 5.2 contrasts the predictions of the two models for *Eclipse*. For *Rhino* we observe a negative net improvement for two of the evaluation metrics and a positive net improvement for two other evaluation metrics.

Overall, in both types of predictions, within and across versions, CK plus LBS are better than CK alone in the majority of the versions. This result is confirmed by almost all average delta percentage values shown next to each net improvement. The average delta percentage

decreased only in 10 out of the 36 metrics computed for the three projects. Hence, we can answer **RQ2** affirmatively.

Table 5.7 Average values when using the CK metrics as independent variables.

Project	Category	Metric	LRM	RF	SVM
ArgoUML	**Rank**	P_{opt}	0.43	0.505	**0.603**
		FPA	28.5	4.91	**45.8**
	Error	E	91.1	88.7	**86.8**
	Classification	A	0.923	0.925	**0.927**
		F	0.0674	**0.199**	0.0812
		MCC	0.0964	**0.199**	0.12
Eclipse	**Rank**	P_{opt}	0.405	0.521	**0.637**
		FPA	51.9	0.444	**60.8**
	Error	E	122	127	**118**
	Classification	A	0.98	0.98	**0.981**
		F	0.0066	**0.0985**	0.0439
		MCC	0.0167	**0.139**	0.104
Rhino	**Rank**	P_{opt}	0.478	0.535	**0.568**
		FPA	17.9	16.3	**21.4**
	Error	E	44.9	42.8	**41.2**
	Classification	A	0.695	0.71	**0.717**
		F	0.544	0.538	**0.579**
		MCC	0.3	0.336	**0.375**

5.2.3 RQ3: Which LBS Help More to Explain Faults?

Table 5.9 shows the ranked LBS according to their contribution for SVM. We also indicate within brackets the median rank across versions. The following observations can be made across the different projects: *Synonym similar* is in the top five most important LBS for all projects. *Inconsistent terms* and *overloaded identifiers* are in the top three for two of the

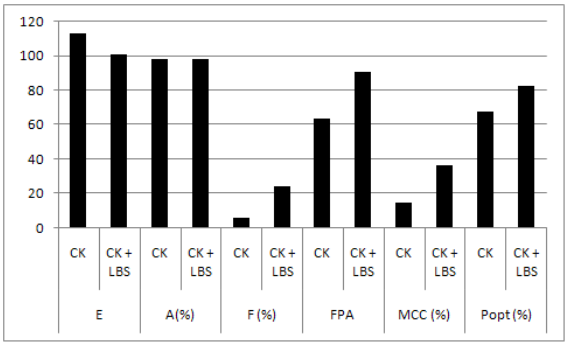


Figure 5.1 Eclipse: Average of the evaluation metrics for same version prediction.

Table 5.8 CK and CK + LBS prediction capability comparison using SVM.

Project	Predi. version	Category	Metric	Imp.	Dec.	Equal	Net imp.	Avg. delta %
ArgoUML	**Same**	**Error**	E	5	1	0	4	-7.495
		Rank	P_{opt}	6	0	0	6	5.068
			FPA	6	0	0	6	10.54
		Classification	A	5	1	0	4	0.5554
			F	5	1	0	4	44.81
			MCC	5	1	0	4	19.13
	Next	**Error**	E	1	3	1	-2	1.883
		Rank	P_{opt}	2	3	0	-1	-0.9093
			FPA	4	1	0	3	12.57
		Classification	A	1	3	1	-2	-0.1643
			F	3	0	2	3	272.4
			MCC	4	0	1	4	285.7
Eclipse	**Same**	**Error**	E	5	0	0	5	-9.369
		Rank	P_{opt}	5	0	0	5	3.177
			FPA	5	0	0	5	12.35
		Classification	A	5	0	0	5	0.1845
			F	5	0	0	5	161.3
			MCC	5	0	0	5	68.52
	Next	**Error**	E	2	2	0	0	2.02
		Rank	P_{opt}	1	3	0	-2	-3.377
			FPA	2	1	1	1	0.8696
		Classification	A	2	2	0	0	-0.03875
			F	4	0	0	4	212.4
			MCC	3	1	0	2	157.1
Rhino	**Same**	**Error**	E	6	1	1	5	-9.596
		Rank	P_{opt}	6	2	0	4	3.144
			FPA	7	0	1	7	4.04
		Classification	A	6	1	1	5	2.012
			F	7	0	1	7	7.895
			MCC	6	1	1	5	11.76
	Next	**Error**	E	1	2	2	-1	2.769
		Rank	P_{opt}	3	2	0	1	3.104
			FPA	4	0	1	4	8.974
		Classification	A	1	2	2	-1	-2.361
			F	2	2	1	0	-0.8509
			MCC	2	2	1	0	-1.667

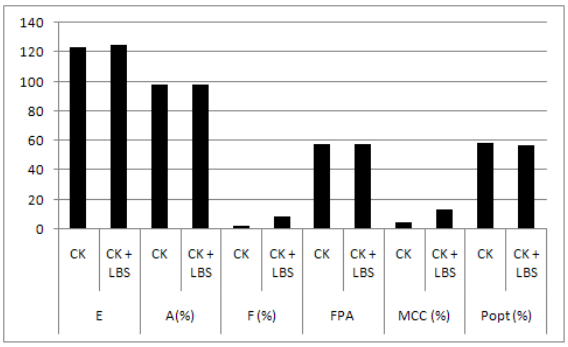


Figure 5.2 Eclipse: Average of the evaluation metrics for next version prediction.

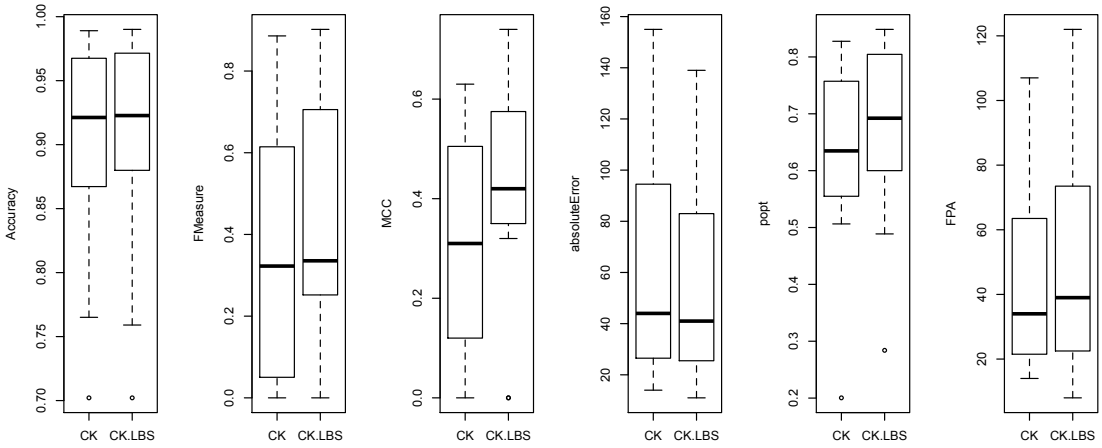


Figure 5.3 All projects: Evaluation metrics for same version prediction.

projects. *Inconsistent terms* and *synonym similar* have a median rank at most 11. Finally, *whole part* does not seem to be important for fault prediction.

We also observe that some LBS tend to have a specific contribution for particular projects. For instance, *extreme contraction* is ranked first among all LBS for *Eclipse*, while *misspelling* is ranked second for *Rhino*.

5.3 Discussions

PCA shows that the majority of the LBS (all but three) are major contributors in at least one dimension for more than 50% of the analyzed versions. The strongest percentages are obtained by *inconsistent terms*, *overloaded identifiers*, and *useless types*. The weakest percentages across versions appear to be *odd grammatical structure*, *misspelling*, and *synonym similar*.

We have analyzed three types of prediction models to identify the best model that works with the CK metrics. Of the analyzed models SVM is found to be the best in the majority of the cases (see Table 5.7). Hence, we have used this model to assess the contribution of LBS to the CK metrics in predicting fault prone classes. The results shown in Table 5.8 indicate that adding LBS to CK improves the predicting capability of the models. The improvement is observed on almost all types of evaluation metrics used in the three projects, considering within and across version prediction. This result is also confirmed by the average delta percentage. Predictions conducted on the same versions using LBS plus CK metrics have shown improvement in more versions compared to predictions on the next version. For example, in *Eclipse* LBS plus CK metrics improved the prediction in all versions (5 of 5), while across versions the improvement is observed in at most half of the versions (2 of 4). The difference can be observed by comparing Figures 5.1 and 5.2.

The results of **RQ3** show that for fault prediction *synonym similar* is in the top five most important LBS. Our findings are consistent with previous research on program identifiers that suggest that identifiers using synonyms lack conciseness and consistency (Lawrie *et al.*, 2006a). Overall, *synonym similar*, *inconsistent terms*, and *overloaded identifiers* seem to be in general the most important LBS for fault prediction. We also observe that other LBS are important but specific to projects, e.g., *extreme contraction* for *Eclipse* and *misspelling* for *Rhino*.

5.3.1 Threats to Validity

Conclusion validity: We do not perform any statistical test, thus threats to conclusion validity are not applicable in this study.

Table 5.9 Ranked LBS according to SVM.

ArgoUML	**Rhino**	**Eclipse**
Synonym similar (4)	Odd grammatical structure (6.5)	Extreme contraction (3)
Inconsistent terms (6.5)	Misspelling (7.5)	Overloaded identifiers (4)
Overloaded identifiers (8.5)	Inconsistent terms (10)	Identifier construction (4)
Identifier construction (9.5)	Synonym similar (11)	Useless types (7)
Odd grammatical Structure (10)	Meaningless terms (12)	Synonym similar (8)
Misspelling (10.5)	Identifier construction (12.5)	Odd grammatical structure (8)
Useless types (13)	Extreme contraction (13)	Meaningless terms (10)
Extreme contraction (15.5)	Overloaded identifiers (14)	Inconsistent terms (11)
Meaningless terms (20)	Useless types (17.5)	misspelling (14)
Whole part (20)	Whole part (20)	Whole part (20)

Internal validity: To identify LBS, we have used a suite of tools that implement general heuristics that can be configured to accommodate some variability. The three projects considered in our study are developed in different environments and hence are influenced by their respective environments. Using one general configuration for all the projects might affect the results. To handle this threat, we manually explored their documentations, when available, and configured the detectors accordingly. The prediction results also depend on the used models and their configurations. We used default configurations or configurations used in other studies. Further tuning of the parameters however could change the rankings of the models. The best model from **RQ2**, SVM, was used with default parameters. Di Martino *et al.* (2011) suggest the use of genetic algorithms to select the parameters for further improvement of the results.

Construct validity: Our study uses the CK metrics considered by Kpodjedo *et al.* (2011) and others as a baseline to investigate the contribution of LBS in predicting fault proneness of a class. In the literature, however, there are other metrics that are proposed to achieve the same goal.

External validity: We consider only three Java projects which limits the generalizability of the study. However, these projects have different domains and different sizes, which limit this threat.

Reliability validity: We use open-source projects whose source code is available. We

attempt to provide all necessary details to replicate the analysis.

5.4 Conclusion

In this study, we have shown that the identifier quality as measured by LBS capture additional information compared to structural measures such as the CK metric suite. In addition, we have shown that in the majority of the cases using LBS with the structural metrics (CK) improves fault prediction. To assess the improvement, we have used different evaluation metrics that address different aspects of the prediction; the improvement is consistent in almost all types of evaluation metrics.

Among all LBS, the most important ones are *overloaded identifiers* and *inconsistent terms*. In the majority of the projects, these are the major contributors of at least one retained principal component; they are also the most important contributors for fault prediction. Moreover, for fault prediction *synonym similar* is always among the top five most important LBS. On the other hand, we believe that other LBS, not included in this list, should not be deemed irrelevant, as they become important for specific projects, e.g., *extreme contraction* and *misspelling*.

CHAPTER 6

LINGUISTIC ANTIPATTERNS (LAS)

Highlight: In Chapters 4 and 5 we provide evidence on the relation between the quality of the lexicon—as measured by HEHCC and LBS—and the quality of the software. In this Chapter, we evaluate the quality of the lexicon in terms of its consistency. We conjecture that the quality of identifiers alone may not be sufficient to identify lexicon flaws and that the inconsistency among identifiers from different sources (name, implementation, and documentation) can be particularly harmful for developers as they may make wrong assumptions about the code behavior or spend unnecessary time and effort to clarify it when understanding source code. Thus, our objective is to define inconsistencies among identifiers from different sources—naming, documentation, and implementation of an entity—that may possibly impair program understanding.

There are many recognized bad practices in software development known as code smells and AntiPatterns (APs) (Brown *et al.*, 1998; Fowler, 1999). They concern poor design or implementation solutions, as for example the *Blob*, also known as *God class*, which is a large and complex class centralizing the behavior of a part of the project and using other classes simply as data holders. Previous studies indicated that APs may affect software comprehensibility (Abbes *et al.*, 2011) and possibly increase change and fault-proneness (Khomh *et al.*, 2012, 2009). From a recent study by Yamashita et Moonen (2013) it is also known that the majority of developers are concerned about code smells.

In this chapter we define of a new family of software antipatterns, named linguistic antipatterns. Software antipatterns—as they are known so far—are opposite to design patterns (Gamma *et al.*, 1994), i.e., they identify "poor" solutions to recurring design problems. For example, Brown's 40 antipatterns describe the most common pitfalls in the software industry (Brown *et al.*, 1998). They are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem, or misusing good solutions, i.e., design patterns. Linguistic antipatterns shift the perspective from source code structure towards its consistency with the lexicon:

> **Linguistic Antipatterns (LAs)** in software projects are recurring poor practices in the naming, documentation, and choice of identifiers in the implementa-

tion of an entity, thus possibly impairing program understanding.

The presence of inconsistencies can be particularly harmful for developers that can make wrong assumptions about the code behavior or spend unnecessary time and effort to clarify it when understanding source code for their purposes. Therefore, highlighting their presence is essential for producing code easy to understand. In other words, our hypothesis is that the quality of the lexicon depends not only on the quality of individual identifiers but also on the consistency among identifiers from different sources (name, implementation, and documentation). Thus, identifying and cataloguing practices that result in inconsistent lexicon will increase developer awareness and thus contributes to the improvement of the lexicon.

An example of LA, which we have named *"Not answered question"*, is the public method `isClassPathCorrect` of class `ProblemReporter` found in the *Eclipse*[1] project. One would expect that such a method returns a Boolean; instead, the method does not return any value, it sets an attribute and then calls another method to perform the task. It is left to the reader to search for a means to obtain the result. Another example of LA—of type *"Method name and return type are opposite"*, found in the *BWAPI*[2] project—is method `player_enemy_impl` of class `Player_Ally`, which returns a `Player_Ally` object. On the one hand, the class name `Player_Ally` suggests that such a class models a player when she is in the ally state, hence implementing part of a *state* design pattern (Gamma *et al.*, 1994). On the other hand, an outsider may wonder why a method `player_enemy_impl` returns a `Player_Ally` and not for example `Player_Enemy`. In the end, it turns out that the class `Player_Ally` models a player in both "enemy" and "ally" states, i.e., the *state* design pattern is not used.

6.1 Catalog

We defined LAs and group them into categories based on a close inspection of source code examples. We analyzed source code from three open-source Java projects—*ArgoUML*, *Cocoon*, and *Eclipse*. We randomly sampled several files and analyzed the source code looking for examples of inconsistencies of lexicon among different sources of identifiers—i.e., identifiers from the name, documentation, and implementation of an entity. For each file, we analyzed the declared entities (methods and attributes) by asking ourselves questions such as "Is the name of the method consistent with its return type?", "Is the attribute comment consistent with its name?". The set of inconsistencies examples that we found were then organized into an initial set of LAs. We iterated several times over the sampling and coding process and refine the questions based on the newly discovered examples. For example, "What is an inconsistent name for a boolean return type?", "Is `void` a consistent type for method `isValid`?",

1. `http://www.eclipse.org`
2. `https://code.google.com/p/bwapi/`

"What other types are inconsistent with method `isValid`?", etc. Over the iterations LAs were further refined, compared, and grouped into categories; categories are modified according to the new examples of inconsistencies—e.g., some categories are combined, refined, or split to account for the newly defined LAs. As the goal is to capture as many different lexicon inconsistencies as possible, the sampling was guided by the theory—i.e., theoretical sampling (Strauss, 1987)—and thus not representative of the entire population of source code entities. We stopped iterating over the sampling and coding process when new examples of inconsistencies did not anymore modify the defined LAs and their categories. However, during this process we did not follow a thorough grounded-theory approach (Strauss, 1987; Glaser, 1992)—e.g., we did not measure the inter-agreement at each iteration—as the process was meant to identify possible inconsistencies for which we would gather developers' perceptions. Thus, the agreement between the authors of this paper was a guidance rather than a requirement.

The above process resulted in 17 types of LAs, grouped into six categories, three regarding behavior—i.e., methods—and three regarding state—i.e., attributes. For methods, LAs are categorized into methods that (A) "do more than they say", (B) "say more than they do", and (C) "do the opposite than they say". Similarly, the categories for attributes are (D) "the name says more than the entity contains", (E) "the name says less than the entity contains", and (F) "the name says the opposite than the entity contains".

The rest of this section details each LA by providing the rationale behind it, an example coming from real software projects, its consequences, and an example solution.

A.1 - "Get" - more than an accessor

A getter that performs actions other than returning the corresponding attribute without documenting it.

Rationale In Java, accessor methods, also called getters, provide a way to access class attributes. As such, it is not common that getters perform actions other than returning the corresponding attribute except for common practices—e.g., when implementing a lazy initialization (Gamma *et al.*, 1994). Any other action should be documented, possibly naming the method differently than `getSomething`.

Example Method `getImageData` which, no matter the attribute value, every time returns a new object (Figure 6.1).

Consequences The usage of such getters would cause an unexpected allocation of new objects (which normally does not happen with getters), or returning a null value when this should not be the case, i.e., the attribute is not null.

Example solution When additional actions in accessor methods are necessary they need to be documented except for common practices—e.g., when implementing a lazy initialization. A possible solution for the example shown in Figure 6.1 is to rename the method to `createImageData` and to comment the unusual behavior: *"A new ImageData object is created and assigned to the attribute every time the method is called".*

```
public ImageData getImageData() {
    final Point size = this.getSize();
    final RGB black = new RGB(0, 0, 0);
    final RGB[] rgbs = new RGB[256];
    rgbs[0] = black; // transparency
    rgbs[1] = black; // black
    final PaletteData dataPalette =
         new PaletteData(rgbs);
    this.imageData =
         new ImageData(size.x, size.y, 8, dataPalette);
    this.imageData.transparentPixel = 0;
    this.drawCompositeImage(size.x, size.y);
    for (int i = 0; i < rgbs.length; i++) {
        if (rgbs[i] == null) {
            rgbs[i] = black;
        }
    }
    return this.imageData;
}
```

Figure 6.1 *"Get" - more than an accessor (A.1).*

A.2 - "Is" returns more than a Boolean

Method name is a predicate, whereas the return type is not Boolean but a more complex type allowing a wider range of values.

Rationale When a method name starts with the term "is" one would expect Boolean as return type, thus having two possible values for the predicate, i.e., `true` and `false`. Thus, having an "is" method that does not return Boolean, but returns more information is counterintuitive. In such cases, the method should be renamed or, at least, details about the return values should be included in the method comments.

Example Method `isValid` with return type `int` (see Figure 6.2).

Consequences Some of problems related to such LA will be detected at compile time (or even by the IDE), however the misleading naming can still cause misunderstanding on the maintainers' side. However, in programs written in C++, no compilation problem will occur when the return type is `int`.

Example solution When the return type cannot be changed to Boolean, we recommend to rename the method and–or document the return values. An example of documentation for the method shown in Figure 6.2, is *"The method returns* -1 *for 'invalid',* 1 *for 'valid', and* 0 *for 'don't know' ".*

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

Figure 6.2 *"Is" returns more than a Boolean (A.2).*

A.3 - "Set" method returns

A set method having a return type different than `void` and not documenting the return type/values with an appropriate comment.

Rationale Modifier methods, also called setters, are methods that allow assigning a value to a class attribute (the attribute being normally protected or private, hence not directly accessible from outside). By convention, setters do not return anything. More generally, the same statement is valid for methods whose name starts with "set". Thus, a set method having a return type different than void should document the return type/values to avoid any misuse.

Example Method `setBreadth` with return type `Dimension` in Figure 6.3 shows one such case where the method always creates a new object and returns it. A proper documentation would include details about the return values.

Consequences One could use the setter method without storing/checking its returned value, hence useful information—e.g., related to erroneous or unexpected behavior—is not captured.

Example solution The example shown in Figure 6.3 can be improved by documenting that *"The method creates a* `Dimension` *and set its breadth to the value of* `source`." and by renaming the method to `createDimensionWithBreadth`.

```
public Dimension setBreadth(final Dimension target, final int source) {
    if (this.orientation == Orientation.VERTICAL) {
        return new Dimension(source, (int) target.getHeight());
    } else {
        return new Dimension((int) target.getWidth(), source);
    }
}
```

Figure 6.3 *"Set" method returns (A.3).*

A.4 - Expecting but not getting a single instance

Method name indicates that a single object is returned but the return type is a collection.

Rationale When a method name indicates that a single object is returned, one would expect that a single object is also returned. If, instead, the return type is a collection, the method shall be renamed or appropriate documentation is needed.

Example Method `getExpansion` returning `List` (see Figure 6.4)—defined in class `DrillFrame`—suggests that an object `Expansion` will be returned whereas a collection is.

Consequences Although this would unlikely cause faults at run-time, it might cause false expectancies to the developers. When reading `getExpansion`, one would expect to handle a simple object, whereas it is necessary to deal with multiple objects, which requires different source code to analyze the result, e.g., iterators.

Example solution A possible solution for the method shown in Figure 6.4 would be to rename it to `getTreeNodes` and rename the attribute accordingly.

```
/**
 * Returns the expansion state for a tree.
 *
 * @return the expansion state for a tree
 */
public List getExpansion() {
    return this.fExpansion;
}
```

Figure 6.4 *Expecting but not getting a single instance (A.4).*

B.1 - Not implemented condition

The method' comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.

Rationale A leading comment summarizes the behaviour of a method at a higher level of abstraction. It allows developers to grasp the intent of the method and the main lines of the implementation without the need to go over all statements in the method's body. Thus, when a condition is expressed in a method's comment one assumes that the condition is implemented.

Example Figure 6.5 shows a method defined in class `FileEditionEditorInput` that based on the comment *"returns an empty array if this object has no children"* whereas the implementation always returns the same value.

Consequences This LA can have two main consequences. First, clients of the corresponding methods assume the documented behavior resulting in wrong system behavior. Second, during testing—especially black box testing—the tester would invest time and effort to generate test cases for the different conditions, while one test case will cover all method statements (or, in general, less test cases are needed).

Example solution The method shown in Figure 6.5 could document the default behavior if it is intentional: *"This method provides a default behavior by always returning an empty array."*

```
/**
 * Returns the children of this object. When this object is
 * displayed in a tree, the returned objects will be this
 * element's children. Returns an empty array if this object
 * has no children.
 *
 * @param object The object to get the children for.
 */
public Object[] getChildren(final Object o) {
    return new Object[0];
}
```

Figure 6.5 *Not implemented condition (B.1).*

B.2 - Validation method does not confirm

A validation method that neither provides a return value informing whether the validation was successful, nor it documents how to proceed to understand.

Rationale A validation method—i.e., a method whose name starts with, for example, "validate", "check", or "ensure"—is a method performing a check for validity that is usually required as a precondition for other operations. As such, validation methods are expected to inform the user whether the check is successful or not either by returning true/false or by throwing an exception is case the validation fails.

Example Figure 6.6 shows method `checkCollision` defined in class `UMLComboBoxEntry` that neither returns Boolean nor throws an exception.

Consequences One may not know how to handle the outcome of the validation. Very likely, such an outcome is stored somewhere—e.g., an instance variable—however this is not clear from the method specification/-documentation.

Example solution Validation methods must inform the user of the result of the validation by means of return value, exceptions, warnings, or errors. A solution for the example in Figure 6.6 would be to return the variable `collision`. In addition, the actions in case of collision could be extracted in a separate method named `resolveCollision`.

```
public void checkCollision(final String before,
                           final String after) {
    final boolean collision = before != null
            && before.equals(this._shortName) || after != null
            && after.equals(this._shortName);
    if (collision) {
        if (this._longName == null) {
            this._longName = this.getLongName();
        }
        this._displayName = this._longName;
    }
}
```

Figure 6.6 *Validation method does not confirm (B.2).*

B.3 - "Get" method does not return

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 is stored and how to obtain it.

Rationale A method whose name starts with "get" or "return" suggests that an object will be returned as a result of the method execution. Thus, having such methods returning `void` without documenting where the result is stored is counterintuitive.

Example The example in Figure 6.7 shows the source code of a method named `getMethodBodies`, defined in class `Compiler`, which suggests method bodies as result, however nothing is returned.

Consequences One would expect to be able to assign the method return value to a variable. However, since this is not possible, one has to further understand the code to determine where the retrieved data is stored and how to obtain it.

Example solution The example shown in Figure 6.7 could be resolved by renaming the method to `fillMethodBodies` or by adding a documentation: *"The method parses the method bodies and stores the result in the parameter* `unit`*"*.

```
protected void getMethodBodies(
        final CompilationUnitDeclaration unit,
        final int place) {
    // fill the methods bodies in order for the code
    // to be generated
    if (unit.ignoreMethodBodies) {
        unit.ignoreFurtherInvestigation = true;
        return; // if initial diet parse did not work,
                // no need to dig into method bodies.
    }
    if (place < this.parseThreshold) {
        return; // work already done ...
    }
    // real parse of the method....
    this.parser.scanner
            .setSourceBuffer(
                    unit.compilationResult.compilationUnit
                    .getContents());
    if (unit.types != null) {
        for (int i = unit.types.length; --i >= 0;) {
            unit.types[i].parseMethod(this.parser, unit);
        }
    }
}
```

Figure 6.7 *"Get" method does not return (B.3).*

B.4 - Not answered question

The method name is in the form of predicate, whereas nothing is returned.

Rationale A method whose name is a predicate (e.g., starts with "is", "has") is expected to have Boolean as return type where the returned value indicates an assertion or a denial.

Example Figure 6.8 shows an example of method `isValid`, declared in class `ISelectionValidator`, where the name suggests a Boolean value as result but nothing is returned.

Consequences Consequences are similar to those of *"Get" method does not return.* In this case, the developer would even expect to use the method within a conditional control structure, which is however not possible.

Example solution The example shown in Figure 6.8 can be resolved by documenting that *"the result of the validation and the validation message are stored in* `res`*"*, by changing the return type to Boolean, and by returning `true` when the selection is valid and `false` otherwise.

```
public void isValid(final Object[] selection,
        final StatusInfo res) {
    // only single selection
    if (selection.length == 1
            && selection[0] instanceof IFile) {
        res.setOK();
    } else {
        res.setError(""); //$NON-NLS-1$
    }
}
```

Figure 6.8 *Not answered question (B.4).*

B.5 - Transform method does not return

The method name suggests the transformation of an object, however there is no return value and it is not clear from the documentation where the result is stored.

Rationale A method whose name suggests the transformation of an object is expected to return the result or, if this is not the case, document where the results is stored—e.g., if one of the parameter stores the result then this must be clear from its name/documentation.

Example An example of this LA is shown in Figure 6.9—method `javaToNative` defined in class `LocalSelectionTransfer`—where the name suggests a transformation of an object but it is unclear where the result is stored and how to retrieve it.

Consequences Similar to *"Get" method does not return.* Specifically, here one would expect to be able to assign the result of the method to a variable suggested by the method name (*Native* in our example, i.e., a platform-specific representation).

Example solution The example shown in Figure 6.9 could document that *"The result of the conversion is stored in* `transferData`*"* or simply inherit the documentation of the overridden method—as in this case it exists.

```
public void javaToNative(final Object object,
        final TransferData transferData) {
    final byte[] check =
            LocalSelectionTransfer.TYPE_NAME.getBytes();
    super.javaToNative(check, transferData);
}
```

Figure 6.9 *Transform method does not return (B.5).*

B.6 - Expecting but not getting a collection

The method name suggests that a collection should be returned, but a single object or nothing is returned.

Rationale A method whose name suggests that a collection is returned is expected to also have a collection as return type. If the method returns a single object then it must be clear form the documentation what is the implicit aggregation function and the method must be considered for renaming.

Example In the example shown in Figure 6.10, the name of the method, defined in class *SAXParserBase*, suggests that some statistics will be returned, while the method only returns a Boolean value.

Consequences A developer would likely expect that the method will return a set of values (e.g., a time series of temperature, or an array of monitoring data), suggesting that appropriate patterns, such as iterators, are needed to navigate the data structure. Instead, in some cases, the method may return only one of these values, or, in other cases, like the one in Figure 6.10, the returned value is completely inconsistent with the method name.

Example solution A solution for the example shown in Figure 6.10 would be to rename the method to `isStatisticsEnabled` as well as the corresponding attribute.

```
public boolean getStats() {
    return SAXParserBase._stats;
}
```

Figure 6.10 *Expecting but not getting a collection (B.6).*

C.1 - Method name and return type are opposite

The intent of the method suggested by its name is in contradiction with what it returns.

Rationale The name of a method indicates the action that will be performed while its return type specifies the type of the result from this action. As such, the return type mush be consistent, i.e., not in contradiction, with the method's name.

Example The method shown in Figure 6.11, defined in class `ControlEnableState`, is an example of this LA, where the name and return type are inconsistent because the method `disable` returns an "enable" state. With the available documentation, the reader will infer that the return type is a control state that can be enabled or disabled.

Consequences The developers can make wrong assumptions on the returned value and this might not be discovered at compile time. In some cases—e.g., when the method returns a Boolean—the developer could negate (or not) the value where it should not be negated (or it should be).

Example solution To resolve the example shown in Figure 6.11 the class `ControlEnableState` could be renamed to `ControlState` to handle the case where the state is enabled but also where the state is disabled. Thus, the inconsistency with method `disable` is resolved as it will be returning a `ControlState`.

```
/**
 * Saves the current enable/disable state of the given control
 * and its descendents in the returned object; the controls
 * are all disabled.
 *
 * @param w the control
 * @return an object capturing the enable/disable state
 */
public static ControlEnableState disable(Control w) {
    return new ControlEnableState(w);
}
```

Figure 6.11 *Method name and return type are opposite (C.1).*

C.2 - Method signature and comment are opposite

The documentation of a method is in contradiction with its declaration.

Rationale	The leading comment of a method specifies the method's intent at a higher level of abstraction and as such it must be consistent with, i.e., not contradicts, its actual implementation.
Example	Figure 6.12 shows method `isNavigateForwardEnabled` where the name of the method is in contradiction with its comment documenting *"a back navigation"*, as "forward" and "back" are antonyms.
Consequences	Consequences are similar to those of the *Method name and return type are opposite*, and can be even more misleading because the developer is unsure whether to trust the comment or the method's signature. Either the one or the other is outdated or inconsistent, and has to be updated.
Example solution	The inconsistency in the example shown in Figure 6.12 would be resolved by correcting the comment to document *"a forward navigation"* thus being consistent with the implementation.

```
/**
 * Returns true if this listener has a target for a
 * back navigation. Only one listener needs to return
 * true for the back button to be enabled.
 */
public boolean isNavigateForwardEnabled() {
    boolean enabled = false;
    if (this._isForwardEnabled == 1) {
        enabled = true;
    } else {
        if (this._isForwardEnabled != 0) {
            enabled =
                this.navigateForward(false) != null;
        }
    }
    return enabled;
}
```

Figure 6.12 *Method signature and comment are opposite (C.2).*

D.1 - Says one but contains many

An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.

Rationale The name of an attribute indicates what is the object(s) that it contains while the type of an attribute indicates the type of the contained object(s). Thus, there must be a consistency between the name and type, i.e., when the name suggests a single instance the type must also do.

Example In the example shown in Figure 6.13, attribute `target` of type `Vector`, it is unclear whether a change affects one or multiple instances in the collection.

Consequences Lack of understanding of the class state/associations. When such attribute changes, one would not know whether the change impacts a one or multiple objects.

Example solution The inconsistency in the example shown in Figure 6.13 can be resolved by renaming the attribute to `targetCritics` or simply `critics`.

```
Vector _target;
```

Figure 6.13 *Says one but contains many (D.1).*

D.2 - Name suggests Boolean but type does not

An attribute name suggests that its value is *true* or *false*, while its declaring type is not Boolean and the declared type and values are not documented.

Rationale The name of an attribute and its type must be consistent in a way that when the name suggests that a Boolean value is contained then the declared type must be indeed Boolean.

Example Figure 6.14 shows one such case defined in class `ExceptionHandlingFlowContext`. The attribute name—`isReached`—suggests that the value will be true if something is reached, false otherwise. However, the declaring type is not Boolean.

Consequences The developer would expect to be able to test the attribute in a control flow statement condition. However, this is not the case, especially in cases like the one in Figure 6.14, for which the returned type is an array, therefore it is not clear how to handle this attribute.

Example solution To resolve the inconsistency in the example shown in Figure 6.14 the type of the array can be changed to `boolean[]` or a comment should be added to documment how the values are treated, e.g., *"0 indicates 'false', every other value is treated as 'true'.*

```
int[] isReached;
```

Figure 6.14 *Name suggests Boolean but type does not (D.2).*

E.1 - Says many but contains one

Attribute name suggests multiple objects, but its type suggests a single one.

Rationale	The name and type of an attribute must be consistent in a way that when the name suggests multiple objects the type must also do. If this is not the case the documentation mush state the rationale behind such inconsistency or the attribute must be renamed to include the implicit aggregation function.
Example	In the example shown in Figure 6.15, the attribute name, defined in class `SAXParserBase`, suggests that it contains statistics whereas its type is Boolean.
Consequences	Lack of understanding of the impact of attribute changes (see also *Says one but contains many*).
Example solution	A solution for the example shown in Figure 6.15 would be to rename the attribute to `statisticsEnabled`.

```
private static boolean _stats = true;
```

Figure 6.15 *Says many but contains one (E.1).*

F.1 - Attribute name and type are opposite

The name of an attribute is in contradiction with its type as they contain antonyms.

Rationale The name and declaring type of an attribute are expected to be consistent with each other and thus one must not contradict the other.

Example The example of Figure 6.16 shows an attribute of class `ActionNavigability`. The contradiction comes form the use of the antonyms "start" and "end", one being part of the type of the attribute, the other being part of its name.

Consequences This kind of misleading attribute naming can induce wrong assumptions. For example, whether a Boolean attribute contains information that can be used directly in a control flow statement condition, or whether it has to be negated. Similarly, prefixes/suffixes such as "start" and "end" could confuse the developer about the direction a data structure should be traversed.

Example solution One way to resolve to inconsistency in the example shown in Figure 6.16 would be to rename class `MAssociationEnd` to `MAssociationExtremity`. Thus, an object of type `MAssociationExtremity` called `start` would mean that the object is the start of the association and will not cause a confusion.

```
MAssociationEnd start = null;
```

Figure 6.16 *Attribute name and type are opposite (F.1).*

F.2 - Attribute signature and comment are opposite

Attribute declaration is in contradiction with its documentation.

Rationale	The comment of an attribute clarifies its intent and as such there must be no contradiction between the attribute's comment and declaration.
Example	The example in Figure 6.17 shows an attribute named `INCLUDE_NAME_DEFAULT`, defined in class `EncodeURLTransformer`. However, its comment says *"Configuration default exclude pattern"*. Whether the pattern is included or excluded is therefore unclear from the comment and name.
Consequences	A first consequence may be increased comprehension effort as without a deep analysis of the source code, the developer might not clearly understand the role of the attribute. As another risk may be that one simply assumes the intent, i.e., trust the name or the comment, without investigating which of the two is correct.
Example solution	To resolve the inconsistency in Figure 6.17 the comment needs to be corrected to document the *"default include pattern"* thus being consistent with the name of the attribute—i.e., `INCLUDE_NAME_DEFAULT`.

```
/**
 * Configuration default exclude pattern,
 * ie .*\/@href|.*\/@action|frame/@src
 */
public final static String INCLUDE_NAME_DEFAULT
    = ".*/@href=|.*/@action=|frame/@src=";
```

Figure 6.17 *Attribute signature and comment are opposite (F.2).*

6.2 Linguistic AntiPattern Detector (LAPD)

We implemented possible LAs detection algorithms in an offline tool, named LAPD (Linguistic AntiPattern Detector), for Java and C++ source code. LAPD analyzes signatures, leading comments, and implementation of program entities (methods and attributes). It relies on the Stanford natural language parser (Toutanova et Manning, 2000) to identify the POS of the terms constituting the identifiers and comments and to establish relations between those terms. Thus, given the identifier `notVisible`, we are able to identify that 'visible' is an adjective and that it holds a negation relation with the term 'not'.

Finally, to identify semantic relations between terms LAPD uses the WordNet ontology (Miller, 1995). Thus, we are able to identify that 'enemy' and 'ally' are antonyms.

Consider for example the code shown in Figure 6.18. To check whether it contains a LA of type *"Get" - more than an accessor* (A.1) LAPD first analyses the method name. As it follows the naming conventions for accessors—i.e., starts with 'get'—LAPD proceeds and searches for an attribute named `imageData` of type `ImageData` defined in class `CompositeImageDescriptor`. The existence of the attribute indicates that the implementation of `getImageData` would be expected to satisfy the expectations from an accessor, i.e., return the value of the corresponding attribute. Thus LAPD analyses the body of `getImageData` and reports the method as an example of *"Get" - more than an accessor* (A.1) as it contains a number of additional statements before returning the value of `imageData`. Indeed, one can note that the value of the attribute is always overridden (line 69) which is not expected from an accessor except if the value is `null`—as for example the Proxy and Singleton design patterns. Further details regarding the detection algorithms of LAs can be found in Appendix A.

For Java source code, we also made available an LAPD [3] online version integrated into Eclipse as part of the Eclipse Checkstyle Plugin [4]. Checkstyle [5] is a tool helping developers to adhere to coding standards, which are expressed in terms of rules (checks), by reporting violations of those standards. Users may choose among predefined standards, e.g., the *Sun* coding conventions [6], or define their owns. Figure 6.18 shows a snapshot of a code example and an LA, of type *"Get" - more than an accessor* (A.1), reported by the LAPD CHECKSTYLE PLUGIN[3] detected in the example. After analyzing the entity containing the reported LA, the user may decide to resolve the inconsistency or disable the warning report for the particular entity.

6.2.1 Evaluation of the Performances

The *goal* of this study is to investigate the performances of the LAPD, with the *purpose* of understanding to what extend the tooling can impact the study on the relevance on the phenomenon. The *quality focus* is software comprehensibility that can be hindered by LAs. The *perspective* is of researchers interested to develop recommending systems aimed at detecting the presence of LAs and suggesting ways to avoid them. The *context* consists of four Java projects, namely two versions of *ArgoUML*, one version of *Cocoon*, and one version of *Eclipse*. Details regarding the projects can be found in Appendix B. We have chosen projects having different size, and for one of them both an old version and a new one.

3. `http://www.veneraarnaoudova.ca/tools`
4. `http://eclipse-cs.sourceforge.net/`
5. `http://checkstyle.sourceforge.net/`
6. `http://www.oracle.com/technetwork/java/codeconv-138413.html`

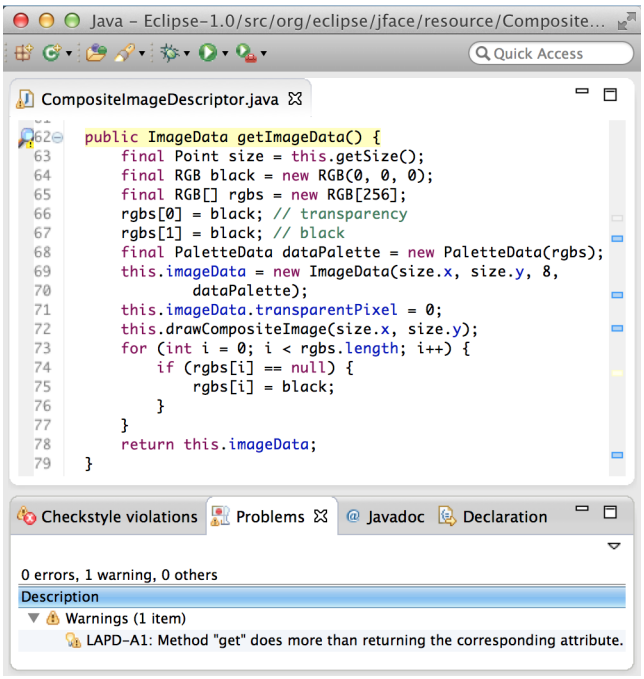


Figure 6.18 LAPD Checkstyle plugin: *"Get" - more than an accessor (A.1).*

The study aims at answering the research question:

RQ1: *How Accurate are the Detected LAs?*

Table 6.1 reports, for each project, the number of detected and validated LAs, as well as the precision of the implemented algorithms. The validated sample for each LA is randomly selected and its size is statistically significant considering a confidence level of 95% and a confidence interval of ±10% (Sheskin, 2007).

Based on the validated sample, LAPD has an average precision of 71% [7] (see Table 6.1). There are two cases in which the precision is below 10% and those are *Attribute signature and comment are opposite* and *Method signature and comment are opposite*. This is due to the difficulty of capturing opposite meaning. An example of the latter is method *close*, defined in class *DeltaProcessor* (Eclipse), with comment *"Closes the given element, which removes it from the cache of open elements"*, which will be detected because of the antonyms "open" and "close".

7. Originally we reported a precision of 72% (Arnaoudova *et al.*, 2013) due to a larger sample for 5 of the LAs.

Table 6.1 Detected LAs.

		ArgoUML 0.10.1	*ArgoUML 0.34*	*Cocoon 2.2.0*	*Eclipse 1.0*	**Validated**	**TP**	**Precision**
A.1	*"Get" - more than an accessor*	0	2	1	15	15/18	9	60%
A.2	*"Is" returns more than a Boolean*	2	0	4	26	24/32	24	100%
A.3	*"Set" method returns*	4	30	6	53	47/93	46	98%
A.4	*Expecting but not getting a single instance*	7	3	8	33	34/51	26	77%
B.1	*Not implemented condition*	20	28	43	232	74/323	58	78 %
B.2	*Validation method does not confirm*	1	8	11	235	70/255	52	74%
B.3	*"Get" method does not return*	1	3	2	57	38/63	37	97%
B.4	*Not answered question*	0	2	0	34	26/36	26	100%
B.5	*Transform method does not return*	0	86	15	44	58/145	57	98%
B.6	*Expecting but not getting a collection*	8	39	12	135	64/194	47	73%
C.1	*Method name and return type are opposite*	0	0	0	6	6/6	3	50%
C.2	*Method signature and comment are opposite*	7	20	12	243	72/282	6	8%
D.1	*Says one but contains many*	15	92	42	103	70/252	40	57%
D.2	*Name suggests Boolean but type does not*	14	13	21	138	64/186	36	56%
E.1	*Says many but contains one*	45	117	24	116	73/302	55	75%
F.1	*Attribute name and type are opposite*	1	0	0	0	1/1	1	100%
F.2	*Attribute signature and comment are opposite*	1	0	3	19	19/23	1	5%

6.3 Relevance of the Phenomenon

The *goal* of this study is to investigate the presence of LAs in software projects, with the *purpose* of understanding the relevance of the phenomenon. The *quality focus* is software comprehensibility that can be hindered by LAs. The *perspective* is of researchers interested to develop recommending systems aimed at detecting the presence of LAs and suggesting ways to avoid them. The *context* consists open-source Java and C++ projects, namely *ArgoUML*, *Cocoon*, *Eclipse*, *Apache Maven*, *Apache OpenMeetings*, *GanttProject*, *boost*, *BWAPI*, *CommitMonitor*, and *OpenCV*. We have chosen projects from various application domains and with different size. Details regarding the projects can be found in Appendix B.

The study aims at answering the research question:

RQ2: *To What Extent do the Analyzed Projects Contain the LAs Defined in Section 6.1?*

Table 6.2 shows the number of detected instances of LAs per kind of LA and per project.

Table 6.3 shows how relevant is the phenomenon in the studied projects. For each LA we report its relevance with respect to the population for which it has been defined as well as its relevance with respect to the total entity population of its kind. For example, the first row of Table 6.3—*"Get" - more than an accessor* (A.1)—shows that such complex accessors represent 2.65% of the accessors and 0.05% of all methods. By looking at the table, the percentage of LAs instances may appear rather low (Min.: 0.02%; 1st Qu.: 0.17%; Median: 0.26%; Mean: 0.61%; 3rd Qu.: 0.65%; Max.: 3.40%). However, in their work on smell detection using change history information Palomba *et al.* (2013) provide statistics about the number of actual classes involved in 5 types of code smells in 8 Java projects; the percentages of affected classes are below 1% for each type of smell, thus somewhat consistent with our findings—although a direct comparison is difficult (due to the different types of entities) the numbers can be taken as a rough indication. Slightly higher are the statistics provided by Moha *et al.* (2010) in which for 10 projects the percentage of affected classes for 4 design smells are as follows: Blob: 2.8%, Functional Decomposition: 1.8%, Spaghetti Code: 5.5%, and Swiss Army Knife: 3.9%.

Moreover, when we consider only the relevant population, the phenomenon appears to be sufficiently important to justify our interest (Min.: 0.02%; 1st Qu.: 1.19%; Median: 5.98%; Mean: 16.89%; 3rd Qu.: 19.33%; Max.: 69.53%).

6.4 Discussions

Based on a close inspection of the source code of several open-source projects, we define a catalog of 17 LAs that we believe are poor practices as they may impede program understanding. Those practices relate to inconsistencies among the name, implementation, and

documentation of source code entities in OO programming languages, in particular methods and attributes. We implement possible detection algorithms to automatically detect LAs and estimate the overall precision of the detection to 71%. Using the detection tool—Linguistic AntiPattern Detector (LAPD)—we study the prevalence of LAs in 11 Java and C++ projects and show that the phenomenon appears sufficiently important to justify our interest.

6.4.1 Threats to Validity

Conclusion validity: Our study is an exploratory study in which we do not make use of statistical tests to reject specific hypotheses. The only issue related to conclusion validity is the representativeness of the sample used to validate the LAa detection precision. We performed for each LA a random sampling across considering a confidence level of 95% and a confidence interval of ±10%.

Internal validity: Threats to internal validity concern the selection of projects. We mitigate this threat by choosing projects different in size, application domain, and programming languages.

Construct validity: In this work, threats to construct validity are mainly due to the mapping between the LA definitions and their detection procedure. In terms of precision, we have mitigated such a threat by manually analyzing a sample of the detected LAs. However, it is worth noting that the detection relies on tools—ontological databases such as WordNet (Miller, 1995) and natural language parsers such as the Stanford Part-of-Speech Analyzer (Toutanova et Manning, 2000)—explicitly conceived to process natural language documents rather than source code. As pointed out by Hindle *et al.* (2011), such tools can be far from optimal when applied to source code.

External validity: Although we cannot really ensure full diversity (Nagappan *et al.*, 2013), the chosen projects are pretty different in terms of size, application domain, and programming languages (Java and C++). As Zhang *et al.* (2013) show, application domain and programming language are the two contexts that could impact source code documentation metrics.

6.5 Conclusion

Previous works measure the quality of the lexicon by the quality of the code identifiers. Our conjecture is that the quality of the identifiers taken in isolation may not be always sufficient to reveal flaws. To prove our conjecture we analyze three open-source Java projects and identify examples of lexicon inconsistencies that we believe may impair program understanding. We abstract the recurring examples into a catalog of Linguistic Antipatterns (LAs),

which we define as recurring inconsistencies in methods/attributes name, implementation, and comments. Thus, the catalogue provides examples of LAs from real projects, illustrated their possible consequences, and outlines possible strategies for their detection. We categorize the catalogue of LAs into six categories as follows:

— *methods*, categorized in cases where a method (i) does more than it says, (ii) says more than it does, and (iii) does the opposite than it says.

— *attributes*, categorized in cases where an attribute (i) contains more than it says (ii) says more than it contains, and (iii) contains the opposite than it says.

In addition, we have carried out a study investigating the presence of LAs in 11 Java/C++ projects. The study, which is based on a first implementation of detector named Linguistic AntiPattern Detector (LAPD), allows us to conclude that LAs are not particular to the project from which we define them and occur sufficiently often to justify our interest.

Table 6.2 LAs : Detected occurrence in the studied projects.

		ArgoUML 0.10.1	*ArgoUML 0.34*	*Cocoon 2.2.0*	*Eclipse 1.0*	*Apache Maven 3.0.5*	*Apache OpenMeetings 2.1.0*	*GanttProject*	*boost 1.53.0*	*BWAPI*	*CommitMonitor 1.8.7.831*	*OpenCV*	Total
A.1	*"Get" - more than an accessor*	0	2	1	15	6	2	2	0	0	1	36	65
A.2	*"Is" returns more than a Boolean*	2	0	4	26	1	0	5	137	2	36	33	246
A.3	*"Set" method returns*	4	30	6	53	314	29	9	6	73	50	67	641
A.4	*Expecting but not getting a single instance*	7	3	8	33	40	78	42	16	0	0	5	232
B.1	*Not implemented condition*	20	28	43	232	2	1	1	1	0	9	3	340
B.2	*Validation method does not confirm*	1	8	11	235	27	1	0	297	4	18	19	621
B.3	*"Get" method does not return*	1	3	2	57	17	5	3	0	0	0	0	88
B.4	*Not answered question*	0	2	0	34	0	0	1	5	0	0	3	45
B.5	*Transform method does not return*	0	86	15	44	1	4	0	46	11	24	177	408
B.6	*Expecting but not getting a collection*	8	39	12	135	14	27	19	12	55	3	16	340
C.1	*Method name and return type are opposite*	0	0	0	6	0	1	2	15	2	0	0	26
C.2	*Method signature and comment are opposite*	7	20	12	243	7	68	8	55	44	288	105	857
D.1	*Says one but contains many*	15	92	42	103	42	31	102	1272	219	47	825	2790
D.2	*Name suggests Boolean but type does not*	14	13	21	138	9	25	11	89	171	151	194	836
E.1	*Says many but contains one*	45	117	24	116	13	7	6	305	77	388	680	1778
F.1	*Attribute name and type are opposite*	1	0	0	0	0	0	2	528	0	5	5	541
F.2	*Attribute signature and comment are opposite*	1	0	3	19	0	1	0	9	3	88	94	218
		126	443	204	1489	493	280	213	2793	661	1108	2262	10072

Table 6.3 LAs : Relevance of the phenomenon in the studied projects.

		Relevance of the phenomenon	Considered population	Relevance with respect to the entities of the same kind
A.1	*"Get" - more than an accessor*	2.65% (65/2457)	getters	0.05% (65/129984)
A.2	*"Is" returns more than a Boolean*	7.44% (246/3307)	methods starting with 'is'	0.19% (246/129984)
A.3	*"Set" method returns*	10.95% (641/5855)	methods starting with 'set'	0.49% (641/129984)
A.4	*Expecting but not getting a single instance*	1.72% (232/13527)	methods expecting single instance to be returned	0.18% (232/129984)
B.1	*Not implemented condition*	6.39% (340/5317)	methods having a documented condition	0.26% (340/129984)
B.2	*Validation method does not confirm*	69.31% (621/896)	validation method	0.48% (621/129984)
B.3	*"Get" method does not return*	0.52% (88/17065)	methods whose name suggest that a result will be returned	0.07% (88/129984)
B.4	*Not answered question*	1.19% (45/3783)	methods whose name suggest Boolean value as a result	0.03% (45/129984)
B.5	*Transform method does not return*	19.33% (408/2111)	transform method	0.31% (408/129984)
B.6	*Expecting but not getting a collection*	23.35% (340/1456)	methods whose name suggest that a collection is returned	0.26% (340/129984)
C.1	*Method name and return type are opposite*	0.02 % (26/129984)	methods	0.02% (26/129984)
C.2	*Method signature and comment are opposite*	2.53% (857/33910)	documented methods	0.66% (857/129984)
D.1	*Says one but contains many*	5.98% (2790/46624)	the number of attributes whose name suggests that it contains a single object	3.41% (2790/81886)
D.2	*Name suggests Boolean but type does not*	64.31% (836/1300)	then number of attributes whose name suggest that it contains a boolean value	1.02% (836/81886)
E.1	*Says many but contains one*	69.53% (1778/2557)	attributes whose names suggest plural	2.17% (1778/81886)
F.1	*Attribute name and type are opposite*	0.66% (541/81886)	attributes	0.66% (541/81886)
F.2	*Attribute signature and comment are opposite*	1.18% (218/18498)	documented attributes	0.27% (218/81886)

CHAPTER 7

LAS: PERCEPTION OF EXTERNAL DEVELOPERS

Highlight: In Chapter 6, we formulated the notion of source code LAs, i.e., recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity and we defined a catalog of 17 types of LAs related to inconsistencies. To understand whether such LAs would be relevant for software developers, we seek answer to the general question: *Do developers perceive LAs as indeed poor practices?* We evaluate the relevance of LAs to developers from the point of view of *external* developers—i.e., not familiar with the code in which the LAs occur.

Although tools may detect instances of (different kinds of) bad practices, they may or may not turn out to be actual problems for developers. For example, by studying the history of projects Raţiu *et al.* (2004) showed that some instances of antipatterns, e.g., *God classes* being persistent and stable during their life, are considered harmless.

This chapter aims at answering the questions stated above, by conducting an empirical study with software developers. In this study, to which we will refer ad *Study I*, we showed to 30 developers an extensive set of code snippets from three open-source projects, some of which containing LAs, while others not. Participants were *external* developers, i.e., people that have not developed the code under investigation, unaware of the notion of LAs. The rationale here is to evaluate how relevant are the inconsistencies, by involving people having no bias—neither with respect to our definition of LAs, nor with respect to the code being analyzed.

7.1 Study Design

The *goal* of the study is to collect opinions about code snippets containing LAs from the *perspective* of external developers, i.e., people new to the code containing LAs—with the *purpose* of gaining insights about developers' perception of LAs. The feedback of external developers will help us to understand how LAs are perceived by developers who are new to the particular code, as it is often the case when developers join a new team or maintain a large project they are not entirely familiar with. Specifically, the study aims at answering the following research questions:

RQ1: *How do External Developers Perceive Code Containing LAs?* We investigate whether developers actually recognize the problem and in such case how important they believe the problem is.

RQ2: *Is the Perception of LAs Impacted by Confounding Factors?* We investigate whether results of **RQ1** depend on participant's i) main programming language (for instance Java versus C++, as the LAs were originally defined for Java), ii) occupation (i.e., professionals or students), and iii) years of programming experience.

In the following, we report details of how the study has been planned and conducted.

7.1.1 Experiment Design

The study was designed as an online questionnaire estimated to take about one hour for an average of two and a half minutes per code snippet. However, participants were told that this time is a simple estimation, that they will be asked questions regarding 25 code snippets, and that there is no actual time limit—i.e., they are free to take all the necessary time to complete the questionnaire. Online questionnaire was preferred over in person interview as it is more convenient for the participants. Participants were free to decide when to fill the questionnaire and in how many steps to complete it—i.e., participants may decide to complete the questionnaire in a single session or to stop in between questions and to resume later. To avoid biasing the participants, we also consider as part of the questionnaire 8 code snippets that do not contain any LA. Thus, we ask participants to analyze 25 code snippets (17 being examples of LAs, and 8 not containing any LA), and to evaluate the quality of each example comparing naming, documentation, and implementation. Ideally, we would have preferred to evaluate an equal number of code snippets with and without LAs. This however would have increased the required time with more than 20 minutes and increased the chances that participants do not complete the survey. Thus, we decided to decrease the number of code snippets without LAs by half (compared to the number of code snippets with LAs).

We selected examples covering the set of LAs from the analyzed projects that in our opinion are representative of the studied LAs. In particular, we used the examples from our previous work as the study was performed before the proceedings were publicly available.

For each code snippet, we formulate a specific question, trying to avoid any researcher bias on whether the practice is good or poor. Note that if the question does not indicate what aspect of the snippet the participants are expected to evaluate, there is a high risk that the participant evaluate an unrelated aspect—i.e., performance or memory related. However, specific questions are subject to the hypothesis guessing bias thus participants may evaluate as poor practices all code snippets as they may guess that this is what is expected. This

is why inserting code snippets that do not contain LAs is a crucial part of the design. To compare the scores given by developers to code snippets that contain LAs and those that do not, we perform a Mann-Whitney test. Thus, for example, when showing the code snippet of method `getImageData` (used in Figure 6.18) corresponding to the example of *"Get" - more than an accessor*, we asked participants to provide their opinion on the practice consisting of *using the word "get" in the name of the method with respect to its implementation*.

To minimize the order/response bias, we created ten versions of the questionnaire where the code snippets appear in a random order. Participants were randomly assigned to a questionnaire. To achieve a design as balanced as possible, i.e., equal number of participants for each questionnaire, we invited participants through multiple iterations. That is, we sent an initial set of invitations to an equal number of participants. After a couple of days we sent a second set of invitations where participants were randomly assigned to the questionnaire that received the lower number of responses.

7.1.2 Objects

For the purpose of the study, we choose to evaluate LA instances detected and manually validated in our previous work (Arnaoudova *et al.*, 2013). Such LAs have been detected in 3 Java software projects, namely *ArgoUML* 0.10.1 and 0.34, *Cocoon* 2.2.0, and *Eclipse* 1.0. Details regarding the projects can be found in Appendix B.

To avoid biasing the participants, we also consider as part of the questionnaire 8 code snippets that do not contain any LA. We ask participants to analyze 25 code snippets (17 being examples of LAs, and 8 not containing any LA), and to evaluate the quality of each example comparing naming, documentation, and implementation.

7.1.3 Participants

Ideally, a target population—i.e., the individuals to whom the survey applies—should be defined as a finite list of all its members. Next, a valid sample is a representative sample of the target population (Shull *et al.*, 2007). When the target population is difficult to define, non-probabilistic sampling is used to identify the sample. In this study, the target population being all software developers, it is impossible to define such list. We selected participants using convenience sampling (Shull *et al.*, 2007). We invited by e-mail 311 developers from open-source and industrial projects, graduate students and researchers from the authors' institutions as well as from other institutions. 31 developers completed the study and after the screen procedure 30 participants remained—11 professionals, and 19 graduate students resulting in a response rate close to 10% as expected Groves *et al.* (2009). Participants were

volunteers and they were not compensated in any way. Anonymity was preserved. Table 7.1 provides information on participants' programming experience and Figure 7.1 shows their native language and the country they live in. Note that the majority of the participants are native French speakers and that only for 13% of the participants are native English speakers. However, we believe that this threat to validity is limited as our questions relate to basic grammar rules (e.g., singular/plural) and we analyze the justification for each question to ensure that the participant understood the question. Note also that the majority of the participants live in Canada.

7.1.4 Study Procedure

We did not introduce participants to the notion of LAs before the study. Instead, we informed them that the task consists of providing their opinion of code snippets.

For each code snippet—containing LAs or not—we asked participants the five questions reported in Table 7.2. With **SI-q1** participants judge the quality of the practice on a 5-point Likert scale (Oppenheim, 1992), ranging between 'Very poor' and 'Very good'. The purpose of **SI-q2** is to ensure that the participants provide their judgement for the practice targeted by the question. In **SI-q3** we are interested to know if participants would undertake an action. We then collect information on the type of action (**SI-q4**) or in case no action would be undertaken the reason why (**SI-q5**). For both **SI-q4** and **SI-q5**, we provide predefined options, to decrease the effort and ease the analysis, however we left space in the form to provide a customized answer. In addition, for each code snippet, we also allow participants to share any additional comment they would make. At any point, participants are free to decide not to answer a question by selecting the option 'No opinion'.

7.1.5 Data Collection

We collected 31 completed questionnaires. Before proceeding with the analysis we applied the following screening procedure: For each LA we remove subjects who chose 'No opinion' as answer to **SI-q1**. The collected answers being in nominal and ordinal scales, standard outlier removal techniques do not apply here.
Thus we first sought for inconsistent answers between questions **SI-q1** and **SI-q3**, i.e., between the quality of the code snippet and whether an action should be undertaken. Although one may judge a code snippet as 'Poor' but believes that no action should be undertaken, we fear that participants providing high number of such combinations may have misunderstood the questions. We intentionally sought for participants providing high number of such combinations ($> 75\%$), resulting in removing one participant.

Table 7.1 Study I - Participants characteristics.

	# of participants	Programming experience (years)	
		< 5	≥ 5
Graduate students	19	9	10
Professionals	11	1	10
Overall	30	10	20

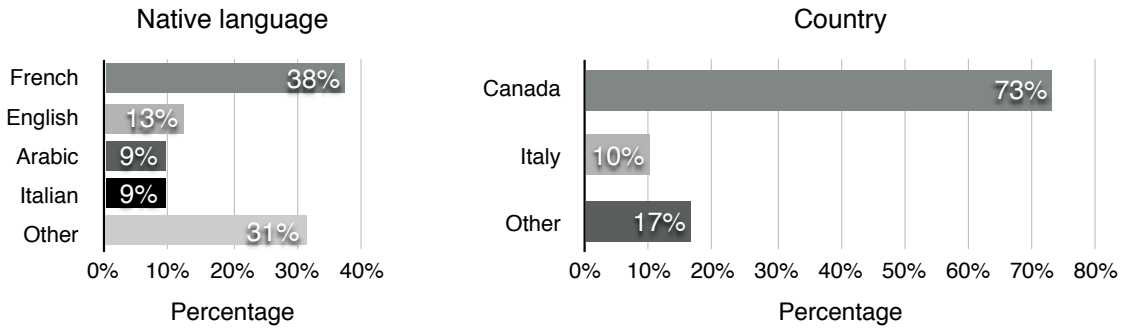


Figure 7.1 Study I—native language and country of the participants.

Then, we individually analyze the justification, i.e., answers of **SI-q2**, and we remove the answer of a participant for an LA if it is clear that the participant judge an aspect different from the one targeted by the LA. For example, when a participants are asked to give their opinion on the use of conditional sentence in comments and no conditional statement in method implementation, participant providing the following justification is removed for the particular LA: *"the method name is well chosen and is well commented too"*. Thus, the number of obtained answers for each kind of LA varies between 25 and 30, as it can be noticed from Figure 7.4.

7.2 Study Results

In the rest of this section we present the results of the study providing both quantitative (Section 7.2.1) and qualitative (Section 7.2.2) analyses.

7.2.1 Quantitative Analysis

Quantitative analysis pertain both **RQs**, i.e., **RQ1** and **RQ2**.

RQ1: How do External Developers Perceive Code Containing LAs?

We first analyzed the developers' perception of examples without LAs. Figure 7.2 shows violin plots depicting the developers' perception of examples without LAs. As expected, those examples are perceived as having a median 'Good' quality (1st quartile: 'Neither good nor poor', median: 'Good', 3rd quartile: 'Very good').

Figure 7.3 shows violin plots depicting the developers' perception of LAs individually for each kind—having a median 'Poor' quality (1st quartile: 'Poor', median: 'Poor', 3rd quartile: 'Neither good nor poor'). Mann-Whitney test indicates that the median score provided to code without LAs is significantly higher than for code with LAs ($p - value$ <0.0001), with a large ($d = 0.66$) Cliff's delta (d) effect size (Grissom et Kim, 2005). Overall, if we consider all LAs, 69% of the participants perceive LAs as 'Poor' or 'Very Poor' practices. However, as Figure 7.3 shows, the perception distribution varies among different LAs. For instance, boxplots—i.e., the inner lines of violin plots—for A.3 (*"Set" method returns*), B.1 (*Not implemented condition*), B.3 (*"Get" method does not return*), B.4 (*Not answered question*), C.2 (*Method signature and comment are opposite*), and F.2 (*Attribute signature and comment are opposite*) have lower quartile at 'Very Poor', median at 'Poor', and, for all of them except B.1, higher quartile at 'Poor'.

We also observe that the perceptions of B.6 (*Expecting but not getting a collection*), D.2 (*Name suggests Boolean but type does not*), E.1 (*Says many but contains one*), and F.1 (*Attribute name and type are opposite*) have little variability and are generally 'Poor'.

On the contrary, the most controversial LAs are A.1 (*"Get" - more than an accessor*) and A.2 (*"Is" returns more than a Boolean*), with lower and higher quartiles being at 'Poor' and 'Good' respectively. Other controversial LAs are A.4 (*Expecting but not getting a single instance*), B.2 (*Validation method does not confirm*), B.5 (*Transform method does not return*), C.1 (*Method name and return type are opposite*), D.1 (*Says one but contains many*), with lower and higher quartiles being at 'Poor' and 'Neither poor nor good' respectively.

In addition to violin plots, we show proportions of the LA perception by grouping, on the one hand, 'Poor' and 'Very Poor' judgements, and on the other hand, 'Good' and 'Very Good' ones. Results are reported in Figure 7.4, where we sort LAs based on the proportion of participants that perceive them as 'Poor' or 'Very Poor'. We observe that, for all but three LAs, majority of participants perceive LAs as 'Poor' or 'Very Poor'. The three exceptions are A.1 (*"Get" - more than an accessor*), A.4 (*Expecting but not getting a single instance*), and D.1 (*Says one but contains many*), for all of which the percentage of participants perceiving them as 'Poor' or 'Very Poor' is 36%, 37%, and 39%, respectively. These are the three LAs having a median perception of 'Neither poor nor good' (see Figure 7.3).

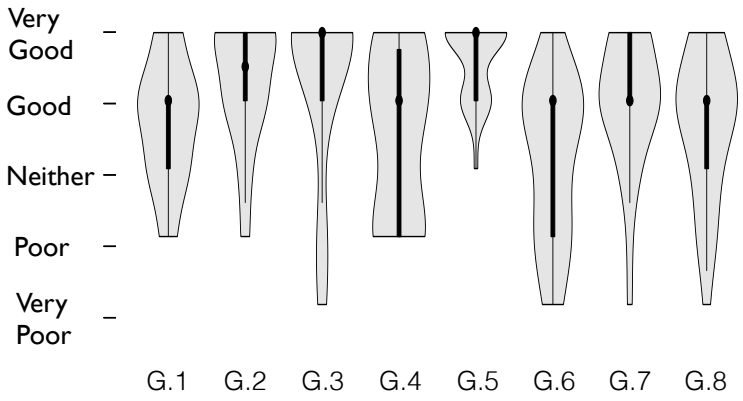


Figure 7.2 Violin plots representing how participants perceive examples without LAs.

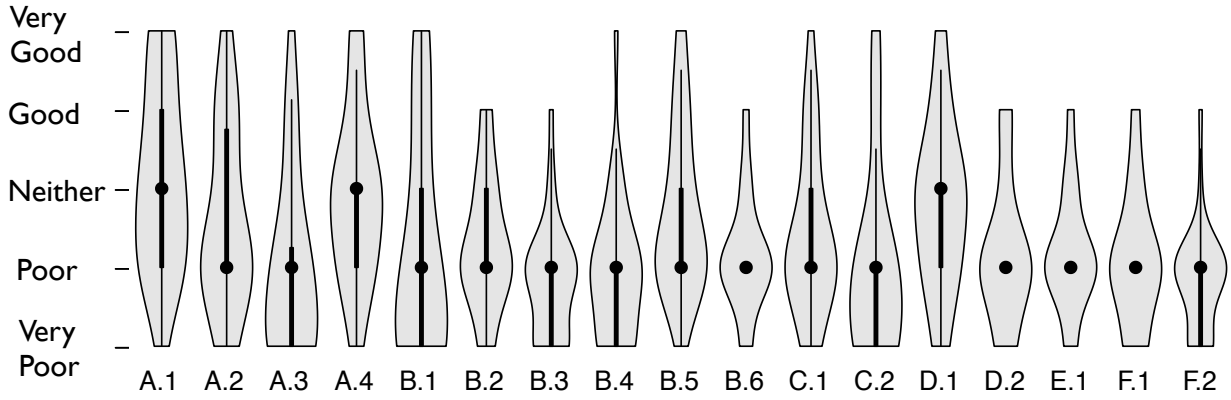


Figure 7.3 Violin plots representing how participants perceive LAs.

RQ2: Is the Perception of LAs Impacted by Confounding Factors?

We grouped the results of the participants according to their (i) main programming language (Java/C# or C/C++), (ii) occupation (student vs. professional), and (iii) years of programming experience (< 5 or ≥ 5 years). The grouping concerning the main programming language is motivated by the different way the languages handle Boolean expressions i.e., in C/C++ an expression returning a non-null or non-zero value is evaluated as true, whereas Java and C# do not perform this cast directly. For this reason, our conjecture is that developers who are used to C/C++ would consider acceptable that a method/attribute that should return/contain a Boolean could instead return/contain an integer.

Main Programming Language: We compared statistically the median perception of participants having C/C++ as main programming language with participants having Java/C# as main programming language. Results of Mann-Whitney test indicate no sig-

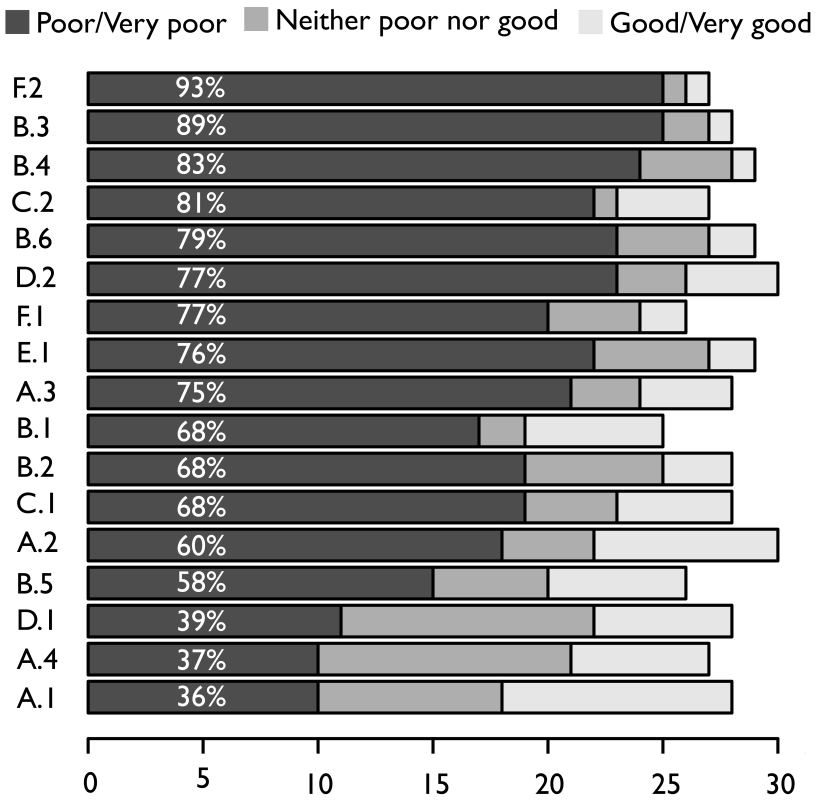


Figure 7.4 Percentage of participants perceiving LAs as 'Poor' or 'Very Poor'.

nificant difference (p-value=0.79) with a negligible Cliff's delta effect size (d=0.01) when all LAs are considered together. We obtained consistent results, i.e., no statistically significant differences, when analyzing each LA separately.

Occupation: By considering all LAs together the difference between the rating given by professionals and students is only marginally significant (p-value=0.06) with a negligible effect size (d=0.10). By considering specific LAs, we found a statistically significant difference only for D.2 (*Name suggests Boolean but type does not*), p-value=0.049, with a medium effect size (d=0.40), and a marginal significance for E.1 (*Says many but contains one*), p-value=0.053, with a medium effect size (-0.39) this time in favor of students).

Experience: Finally, we compared the results between participants having a high experience ($\geq$ 5 years) with others ($<$ 5 years). We found no statistical difference (p-value=0.78) and a negligible (d=-0.11) effect size on the whole data set—i.e., all LAs—as well as when considering each LA separately. Hence, experience does not seem to play a role in the way participants perceive LAs.

Thus, we conclude that developers' perceptions of LAs are not impacted by their main programming language, occupation, or experience.

Next, we provide qualitative analysis on external developers' perception of LAs.

7.2.2 Qualitative Analysis

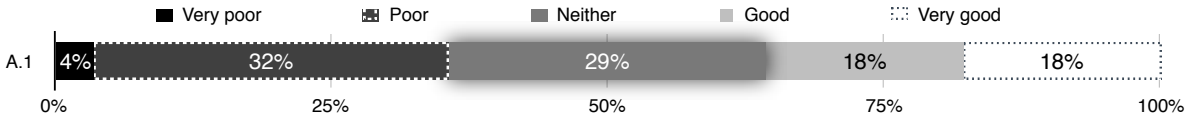
For each type of LA, we briefly summarize its definition and we highlight the perception of external developers.

A.1 - "Get" - more than an accessor

A getter that performs actions other than returning the corresponding attribute without documenting it.

External developers' perception

As shown in the stacked bar chart below, 36% (10 participants), perceived the practice as 'Poor' or 'Very Poor'. 8 participants, i.e., 29%, perceived the practice as 'Neither poor nor good', while they suggested renaming and–or refactoring actions. Finally, 10 participants, i.e., 36%, perceived this practice as 'Good' or 'Very good' *"because this is common practice"*, and, 3 of those commented that the documentation should specify the additional functionality.

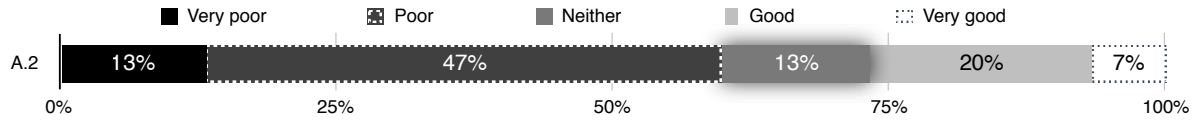


A.2 - "Is" returns more than a Boolean

Method name is a predicate, whereas the return type is not Boolean but a more complex type allowing a wider range of values.

External developers' perception

18 participants (60%)—i.e., the majority—perceived this practice as 'Poor' or 'Very Poor' and would have preferred a Boolean return type. 4 participants (13%) perceived this practice as 'Neither poor nor good'. However, 8 participants (27%) perceived this practice as 'Good' or 'Very Good'. Interestingly, 3 of them explicitly referred to the return values being 0 or 1, and indicated that they are commonly used instead of the Boolean values `false` and `true`. However, the particular method returns -1 (which corresponds to "invalid"), 1 ("valid"), or 0 ("don't know").

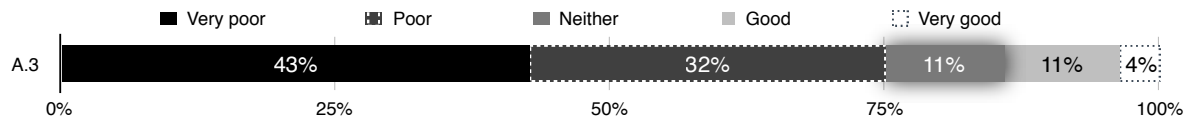


A.3 - "Set" method returns

A set method having a return type different than `void` and not documenting the return type/values with an appropriate comment.

External developers' perception

The majority—21 participants (75%)—perceived this practice as 'Poor' or 'Very Poor' from which 12 participants (43%) perceived this practice as 'Very poor'. 3 participants (11%) perceived this practice as 'Neither poor nor good' and 4 participants (14%) perceived this practice as 'Good' or 'Very Good'. A participant indicated that in OO programming *"majority of coders will agree that the word 'set' is usually used in opposition with 'get' so that many coders will suppose this method is setting a value to a member/attribute. This is a very poor practice since this function is not setting anything but instead creating an object"*. The only participant that perceived this practice as 'Very good' justified that returning a value from a 'set' method can have a benefit as *"in most languages except Java it allows for chaining of method calls"*.

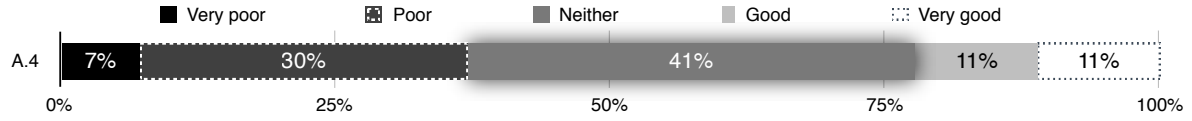


A.4 - Expecting but not getting a single instance

Method name indicates that a single object is returned but the return type is a collection.

External developers' perception

10 participants (37%) perceived this practice as 'Poor' or 'Very Poor'. 11 participants (41%) perceived this LA as 'Neither poor nor good'. 6 of them justified that in the particular case, 'expansion' can be considered as 'list', hence it does not require plural. The other 5 would undertake a renaming. 6 participants (22%) considered this LA as 'Good' or 'Very good', and also justified that 'expansion' suggests a collection, or that they would understand the code by inferring the presence of a collection from the return type or from the comment.

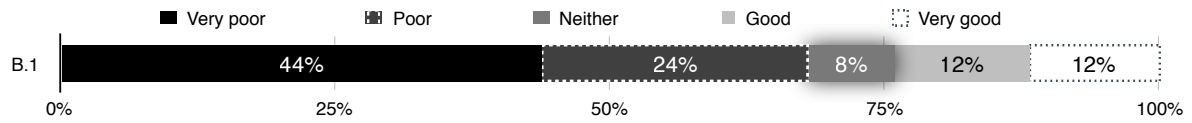


B.1 - Not implemented condition

The method' comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.

External developers' perception

17 participants (68%) perceived this practice as 'Poor' or 'Very Poor'—11 of which (44%) perceived this practice as 'Very poor'. Some of them assumed that the implementation is a placeholder for future code and explained *"...that's really dangerous! Such code builds perfectly and sooner or later will be used by someone who will have a very bad surprise about the results"*. 2 participants (8%) perceived this practice as 'Neither poor nor good' and 6 participants (24%) perceived this practice as 'Good' or 'Very Good'.

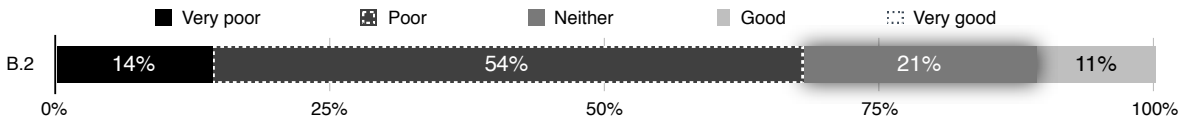


B.2 - Validation method does not confirm

A validation method that neither provides a return value informing whether the validation was successful, nor it documents how to proceed to understand.

External developers' perception

The majority of the participants, i.e., 19 participants (68%), agreed that this is a poor/very poor practice. The remaining 9 participants were more lenient—3 participants (11%) perceived it as 'Good' and 6 participants (21%) as 'Neither poor nor good'. This is mainly because they trust the validation performed by the method, and do not expect a return value. Indeed, one of them explained that it would be better to have a return value informing whether the validation is successful but it is not necessary.

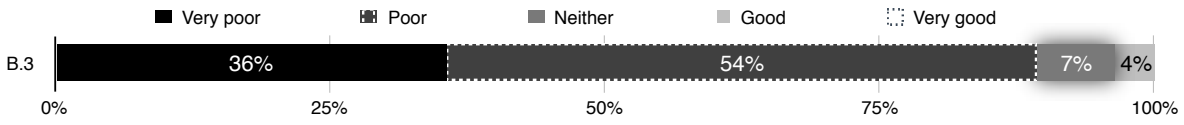


B.3 - "Get" method does not return

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 is stored and how to obtain it.

External developers' perception

25 participants (89%) perceived this practice as 'Poor' or 'Very Poor': all agreed that there should be either a renaming (e.g., 'fill', 'parse', or 'set' instead of 'get') or code modification (e.g., refactoring or changing the return type). 2 participants (7%) perceived this practice as 'Neither poor nor good'; 1 participant (4%) perceived this practice as 'Good'.

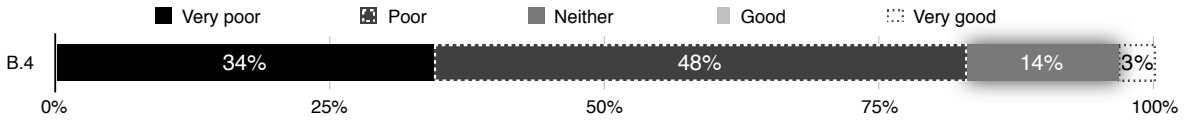


B.4 - Not answered question

The method name is in the form of predicate, whereas nothing is returned.

External developers' perception

24 participants (83%) perceived the practice as 'Poor' or 'Very poor'. Only 4 participants (14%) perceived this practice as 'Neither poor nor good' and 3 of them would undertake an action (renaming or code modification). Only 1 participant (3%) perceived it as 'Very good' because *"it is understandable"*. This participant indicated C as her main programming languages, while being not expert of Java.

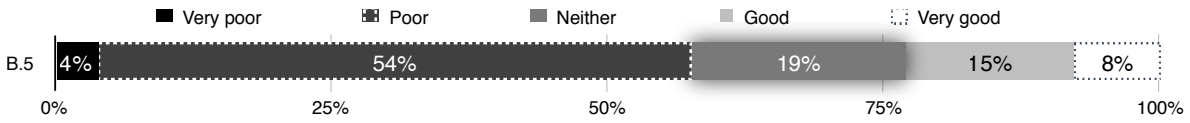


B.5 - Transform method does not return

The method name suggests the transformation of an object, however there is no return value and it is not clear from the documentation where the result is stored.

External developers' perception

15 participants (58%) perceived this practice as 'Poor' or 'Very poor'. 2 participants justified the expected return type by providing as example the `toString` method. From the other 11 participants—5 participants (19%) perceived this practice as 'Neither poor nor good' and 6 participants (23%) perceived this practice as 'Good' or 'Very Good'—2 would prefer to have a (non *void*) return type, although perceiving the practice as 'Neither poor nor good'; and 1 perceived the practice as 'Very good' but justified: *"I would blame for anything the superclass as this is a polymorphic method"*. On the contrary, 3 of the participants explicitly stated that no return type should be expected from a transform method.

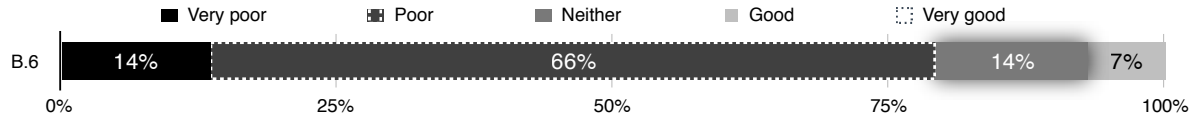


B.6 - Expecting but not getting a collection

The method name suggests that a collection should be returned, but a single object or nothing is returned.

External developers' perception

23 participants (79%) perceived the practice as 'Poor' or 'Very poor'. To reflect the return type, participants suggested a renaming, e.g., `haveStats`, `statsEnabled`, or `statsShown`. From the other 6 participants, 2 (7%) perceived the practice as 'Good', while 4 (14%) as 'Neither good nor poor'. One of these 4 participants justified the choice after wrongly inferring that `stats` stands for 'status', whereas another participant was confused by the Boolean return type.

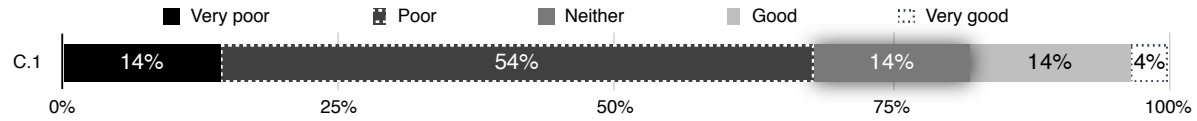


C.1 - Method name and return type are opposite

The intent of the method suggested by its name is in contradiction with what it returns.

External developers' perception

19 participants (68%) perceived it as 'Poor' or 'Very poor'. From the other 9—4 participants (14%) perceived this practice as 'Neither poor nor good' and 5 participants (18%) perceived this practice as 'Good' or 'Very Good'—a participant suggested to rename the return type, to avoid the use of antonyms; another admitted that *"Even if the wording is not totally clear we get that it returns the state"*. The remaining 7 participants had no issue with this practice, and highlighted that the existing comment *"Saves the current enable/disable state ..."* is complementary and clarifies the purpose of the method.

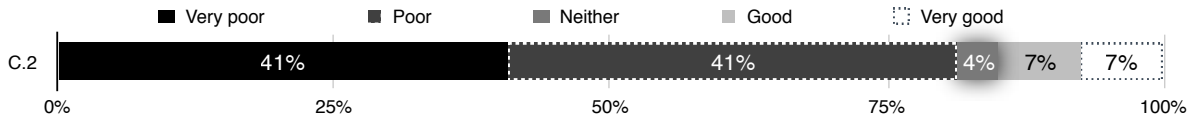


C.2 - Method signature and comment are opposite

The documentation of a method is in contradiction with its declaration.

External developers' perception

22 participants (81%) condemned this practice, with 11 (41%) perceived it as 'Very poor'. One participant explicitly justified that she would trust the naming rather than the comment. This is also reflected by the high percentage (74%) of participants who perceived that the action to be undertaken would be to change the comment. 1 participants (4%) perceived this practice as 'Neither poor nor good'; 4 participants (15%) perceived this practice as 'Good' or 'Very Good'.

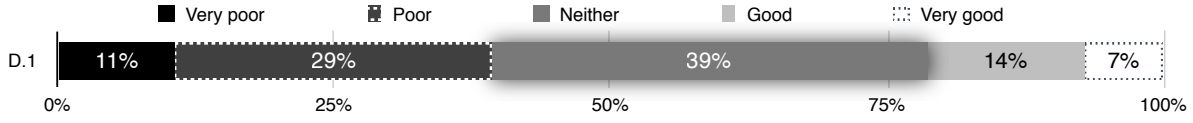


D.1 - Says one but contains many

An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.

External developers' perception

Only 11 participants (39%) perceived this practice as 'Poor' or 'Very poor'. 11 participants (39%) perceived this LA as 'Neither poor nor good', and 7 of them justified their choice to the lack of context. In other words, whether attribute `target` of type `Vector` is a good or poor naming, it depends on whether the target is the entire collection or selected objects contained in the collection. 6 participants (21%) perceived this practice as 'Good' or 'Very good' assuming that target refers to the entire collection.

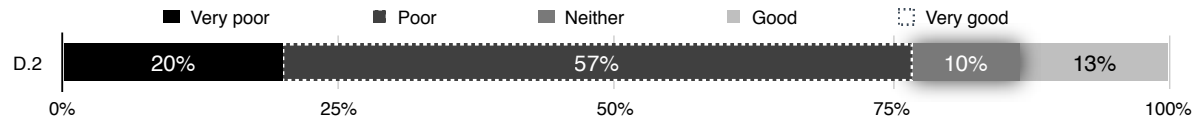


D.2 - Name suggests Boolean but type does not

An attribute name suggests that its value is *true* or *false*, while its declaring type is not Boolean and the declared type and values are not documented.

External developers' perception

23 participants (77%) perceived this practice as at least 'Poor' when we showed them attribute `isReached` of type `int[]`; they expected at least an array of Boolean values. A participant suggested `reachedItems` as a more appropriate name. From the remaining participants, 3 perceived the practice as 'Neither poor nor good' (10%) and 4 as 'Good' (13%) and assumed values are 0 for *false* and 1 for *true*.

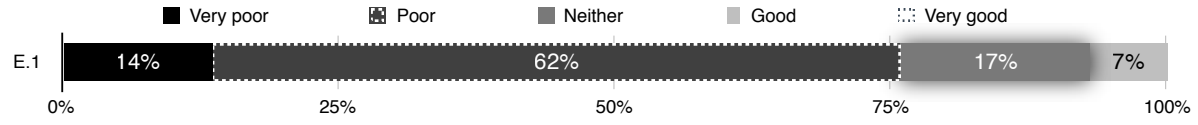


E.1 - Says many but contains one

Attribute name suggests multiple objects, but its type suggests a single one.

External developers' perception

22 participants (76%) perceived this practice as 'Poor' or 'Very poor'. 2 of the remaining 7 participants—5 participants (17%) perceived this practice as 'Neither poor nor good' and 2 participants (7%) perceived this practice as 'Good'—suggested that the attribute is a flag indicating whether statistics are enabled. 2 of them also suggested to add comments to improve understandability.

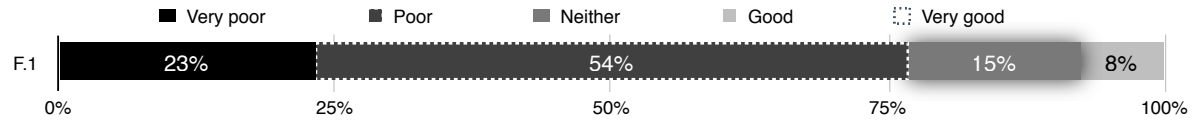


F.1 - Attribute name and type are opposite

The name of an attribute is in contradiction with its type as they contain antonyms.

External developers' perception

20 participants (77%) perceived this practice as 'Poor' or 'Very poor'. From the remaining participants, 4 (15%) of them indicated that this naming may or may not be appropriate, based on the context (thus perceiving it as 'Neither poor nor good'); and 2 (8%) of them perceived this practice as 'Good' and believed that the naming is perfectly legitimate (i.e., it is not confusing to deal with *"starting end and finishing end"*) though one recommended comments to clarify this inconsistency.

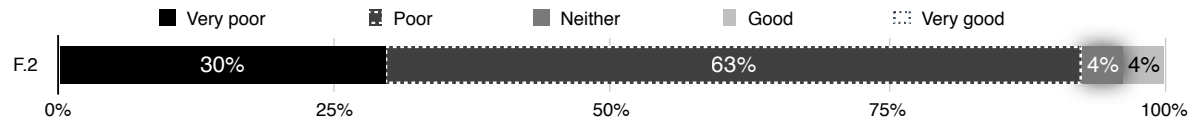


F.2 - Attribute signature and comment are opposite

Attribute declaration is in contradiction with its documentation.

External developers' perception

A large majority of participants,i.e., 25 participants (93%), perceived this practice as 'Poor' or 'Very poor'. One participant commented: *"The most pernicious issue is that most of coders will focus on the meaning of* `.*/@href=|.*/action= |frame /@src=` *(whatever it means) although it is of paramount importance to check the 'exclude/include' property; depending on coders' trend to check first the comment or the name of the member!"*. Another participant commented: *"We don't know who to believe the comments or the attribute name"*. Only 2 participants (7%) were more lenient with their perception—1 participant perceived this practice as 'Neither poor nor good' and 1 participant as 'Good'—one of which commented that one is able to *"get the intent"*.



7.3 Discussions

Overall, the majority of the LAs were perceived as poor or very poor. Some LAs are perceived more severely than others, e.g., *"Set" method returns* (A.3), *Not implemented con-*

dition (B.1), and *Method signature and comment are opposite* (C.2). Developers found less serious LAs where the return type is inconsistent with the method name, e.g., *Method name and return type are opposite* (C.1) and *Attribute name and type are opposite* (F.1). Also, some apparently bad practices in Java come from good or usual practices inherited from other programming languages such as C. The LAs exhibiting the most diverse opinions, i.e., *"Get" - more than an accessor* (A.1) and *"Is" returns more than a Boolean* (A.2) are those for which documentation is crucial, as participants tend to wrongly assume the behavior. We observe that participants are more lenient to some inconsistencies involving the naming and type of an entity, i.e., *Method name and return type are opposite* (C.1) and *Attribute name and type are opposite* (F.1), as compared to the same inconsistencies where the naming and comments of an entity are involved, i.e., *Method signature and comment are opposite* (C.2) and *Attribute signature and comment are opposite* (F.2). Finally, results show that, in general, the main programming language, occupation, and experience of participants did not significantly influence the way developers perceive LAs.

7.3.1 Threats to Validity

Conclusion validity: We use proper non-parametric statistical tests and perform confounding factor analysis.

Construct validity: We manually validated the occurrences of the LAs that we showed to participants, and we selected a sample representative of different kinds of LAs. There is a risk that the collected perception is bound to the particular instances of LAs chosen rather to their category. However, we limited this threat by collecting justifications from the participants that helped us to understand whether the LAs is indeed is a general problem—which we found sometimes to be the case—or whether, instead, it may depend on the particular context.

For what concerns the measurement of the study participants' perception, we used questionnaires expressed in a Likert scale (Oppenheim, 1992), which helps to aggregate and compare results from multiple participants.

Internal validity: When asking participants to evaluate code snippets, we formulated a specific question thus possibly affecting the internal validity of the study as participants may have guessed the expected answer (Shull *et al.*, 2007). To cope with this threat, we also evaluated a set of examples not containing LAs and showed a statistically significant difference in developers' perceptions. We analyzed the effect of experience and of the main programming language used by developers. Another threat to validity is that external developers are only provided with code snippets and thus unaware of the context, i.e., the particular project to which a snippet belongs. Providing context may lead to more lenient evaluations by external developers as they may resolve the inconsistencies from other places in the code (e.g., from

the way the entity is used), which could bias the perception of the practice itself. Also, as participants are external to the project, the lack of domain knowledge may have impacted their perception. We believe that this threat is limited as LAs concern general inconsistencies and thus are domain independent.

External validity: In terms of objects, the two studies have been conducted on four versions of Java projects. Although we cannot really ensure full diversity (Nagappan *et al.*, 2013), the chosen projects are pretty different in terms of size and application domain. In terms of subjects, the studies involved both students and professionals (from industry and from the open-source community).

7.4 Conclusion

In Chapter 6, we defined a catalog of LAs that we believe are poor practices related to inconsistencies among the name, implementation, and documentation of an entity. In this chapter, we showed that code containing such LAs is perceived as poor by the majority (69%) of the developers that we surveyed. Our study involved 30 *external* developers among graduate students and professional, i.e., people that did not participate to the development of the project on which LAs were detected and unaware of the notion of LAs. Overall, participants perceived as more serious the instances when the inconsistency involved both method signature and comments than those involving the method name and return type.

Table 7.2 Study I - Questionnaire.

Question	Possible answers
SI-q1: You judge this practice as:	(Single choice) Very poor Poor Neither poor nor good Good Very good No opinion
SI-q2: Please justify	(Free-form)
SI-q3: Would you undertake an action with respect to the practice?	(Single choice) Change Keep it 'as is' No opinion
SI-q4: Illustrate the kind of action you would undertake (when this is the case).	(Multiple choice) Comments (add/remove/modify) Renaming Implementation (add/remove/modify) Other ___
SI-q5: Explain the reason why you would not undertake any action (when this is the case).	(Multiple choice) It is a common practice Naming and functionality are consistent Comments and naming are consistent Comments and functionality are consistent Other ___

CHAPTER 8

LAS: PERCEPTION OF INTERNAL DEVELOPERS

Highlight: In Chapter 7, we showed that a large majority of the *external* developers perceive LAs as poor practices. However, some questions remain unanswered: *Do developers familiar with the code also perceive LAs as poor practices? If this is the case, would they take any action and remove LAs? What causes LAs to occur?* To answer those questions, we evaluate the relevance of LAs to developers from the point of view of *internal* developers—i.e., familiar with the code in which the LAs occur—and investigate the actions that they would undertake to remove them (if any).

This chapter aims at answering the questions stated above, by conducting an empirical study with software developers who are familiar with the code in which the LAs occur. In this study, to which we will refer as *Study II*, we involved 14 *internal* developers from 8 projects (7 open-source and 1 commercial), with the aim of understanding how they perceive LAs in projects that they know, whether they would remove them, how (if this is the case), and what caused LAs to occur at the first place. Here, we first introduce to developers the definition of the specific LA under scrutiny, after which they provide their perception about examples of LAs detected in their project.

8.1 Study Design

The *goal* of this study is to investigate the perception of LAs from the *perspective* of *internal* developers, i.e., those contributing to the project in which LAs occur. Internal developers will provide us not only with their opinion about LAs but also with insights on the typical actions they are willing to undertake, to correct the existing inconsistencies and possibly help us to understand what causes LAs to occur. The *context* consists of examples of code, selected from projects to which the surveyed developers contribute. To extend the external validity of the results, for this study, we considered projects written in different languages, Java and C++. The study aims at answering the following research questions:

RQ1: *How do Internal Developers Perceive LAs?* This research question is similar to RQ1 of *Study I*, however here we are interested in the perception of developers familiar with the code containing LAs, i.e., of people who contributed to it.

RQ2: *What are the Typical Actions to Resolve LAs?* Other than the opinion on the practices described by LAs, we investigate whether developers are willing to undertake actions to correct the suggested inconsistencies.

RQ3: *What Causes LAs to Occur?* We are interested to understand under what circumstances LAs appear to better cope with them.

8.1.1 Experiment Design

The study was designed as an online questionnaire. The number of LAs was selected so that the questionnaire requires approximately 15 minutes to be completed, and therefore ensures a good responsiveness from internal developers. As in *Study I*, the time was simply indicative, i.e., participants are free to take all the necessary time to complete the questionnaire. As LAs were related to methods having different size and complexity, the questionnaires contained between 5 and 6 examples, i.e., not always the same number. Thus, each participant evaluates only a subset of the LAs. We selected examples of LAs from the analyzed projects that in our opinion are representative of the studied LAs. We selected the examples in a way to have higher diversity, i.e., so that the study includes examples of all 17 types of LAs. Note that in the study with external developers (see Chapter 7) each questionnaire contained examples of all LAs as it was designed to take more time.

8.1.2 Objects

To select the projects for this study we also used convenience sampling. We consider LAs extracted from 8 software projects, specifically 1 industrial, closed-source project, namely *MagicPlan*, and 7 open-source projects. The projects have different size and belong to different domain, ranging from utilities for developers/project managers (e.g., *Apache OpenMeetings*, *GanttProject*, *commitMonitor*, *Apache Maven*) to APIs (*Boost*, *BWAPI*, and *OpenCV*) or mobile applications (*MagicPlan*). Details regarding the projects can be found in Appendix B. We chose more projects than in the study with external developers (Chapter 7), in order to obtain a larger external validity from developers belonging to different projects (including a commercial one), and in order to consider both Java and C++ code.

8.1.3 Participants

The study involved 14 developers from the projects mentioned above. As for the distribution across projects, one developer per project participated in the study, except for *Boost*, for which 3 developers participated, and for *BWAPI*, for which 4 developers participated. Such 14 developers are the respondents from an initial set of 50 ones we invited to participate. Invited

participants were committers whose e-mails were available in the version control repository of the project. Participants were volunteers and they were not compensated. Anonymity was preserved.

8.1.4 Study Procedure

We showed to participants examples of LAs detected in the project they contribute to. As the goal of this work is to evaluate developers' perception of LAs we did not re-evaluate the precision but rather manually validated a subset of the detected examples to assure that they are indeed representative of LAs. The number of LAs was selected so that the questionnaire requires approximatively 15 minutes to be completed, and therefore ensures a good responsiveness from internal developers. As LAs were related to methods having different size and complexity, the questionnaires contained between 5 and 6 examples, i.e., not always the same number.

We selected examples of LAs from the analyzed projects in a way to have higher diversity. The study included examples of all 17 types of LAs. For each example, we first provided participants with the definition of the corresponding LA, and then we asked them to provide an opinion about the general practice—i.e., question **SII-q0** "How do you consider the practice described by the above Linguistic Antipattern?"—using, again, a 5-point Likert scale. Then, we asked participants to provide indications about the specific instance of LA by asking the questions shown in Table 8.1.

8.1.5 Data Collection

We collected responses of 14 developers regarding 47 unique examples of all types of LAs except C.2[1]. The collected answers represent 72 data points, where each data point is a unique combination of a particular example (instance) of an LA and a developer who evaluated it.

8.2 Study Results

In this section we present the results of the study with internal developers and provide both quantitative (Section 8.2.1) and qualitative (Section 8.2.2) analyses.

1. None of the questionnaires containing examples of type C.2 was answered.

Table 8.1 Study II - Questionnaire.

Question	Possible answers
SII-q1: How familiar are you with this code?	(Single choice) I wrote it I didn't write it but I came across this code Don't remember seeing it before Other ___
SII-q2: Why the inconsistency occurred, i.e., what are the causes?	(Multiple choice) Evolution (it was consistent initially) Didn't give it enough thought initially Copy/paste and forgot to change Reuse without changing since it is working Other ___
SII-q3: Equivalent to **SI-q3**	Equivalent to **SI-q3**
SII-q4: Equivalent to **SI-q4**	(Free-form)
SII-q5: Equivalent to **SI-q5**	Equivalent to **SI-q5**

8.2.1 Quantitative Analysis

RQ1: How do Internal Developers Perceive LAs?

Regarding the general opinion of participants (i.e., answers of **SII-q0**), 51% of the times participants perceived LAs as 'Poor' or 'Very poor'. This percentage is lower than the one obtained in *Study I* with external developers, i.e., 69%. In our understanding—and also according to what we observed from developers' comments (see Section 8.2.2)—such a decrease in the proportion may sometimes be due to the context in which LAs occur where internal developers perceive LAs as acceptable.

RQ2: What are the Typical Actions to Resolve LAs?

Participants would undertake an action in 56% of the cases, and in 44% of the cases they believe that the code should be left 'as is'. We discuss the reasons behind these two choices—as reported by the participants—and illustrate them with examples in Section 8.2.2 [2].

Overall, the kind of changes that participants are willing to undertake to reduce the effect of LAs fall into one of the following (or a combination of those) categories: renaming, change [3]

2. We do not report project names with the examples to avoid disclosing the confidentiality of the provided answers.

3. A change may be one or more of the following: modification, addition, or removal.

in comments, and change in implementation. In 42% of the cases, the solution involved renaming, 14% involved a change of comments, and 11% a change in the implementation.

10% (5 out of 47) of the LAs shown to internal developers during the study have been removed in the corresponding projects after we pointed them out. The removed examples were instances of A.2 (*"Is" returns more than a Boolean*), A.3 (*"Set" method returns*), B.2 (*Validation method does not confirm*), and B.4 (*Not answered question*). We report the examples and how they were removed in the corresponding LA tables when discussing the perception of internal developers.

RQ3: What Causes LAs to Occur?

Note that *internal* developers may not be familiar with the whole project to which they contribute and thus they may not be familiar with the examples that we show them. Thus, regarding the possible causes of LAs, we limit our analysis only to cases where the participants wrote the code containing the LAs and cases where they were knowledgeable of that code, e.g., because they were maintaining it. The reported causes and the number of times they occur are as follows (ordered by decreasing order of frequency):

1. *Evolution* (8): The code was initially consistent, but at some point an inconsistency was introduced, hence causing the LA.
2. *Developers' decision* (7): It is a design choice or simply personal preference.
3. *Not enough thought* (5): Developers did not carefully choose the naming when writing the code.
4. *Reuse* (2): Code was reused from elsewhere without properly adapting the naming.

8.2.2 Qualitative Analysis

A.1 - "Get" - more than an accessor

A getter that performs actions other than returning the corresponding attribute without documenting it.

Internal developers

Perception	Developers decided not to refactor the examples of this type. For instance, regarding method `getPhases` which retrieves a result rather than being a simple accessor, one of the developers commented on the decision not to change it: *"perhaps the method could be renamed to `findPhasesForLifecycle`, but if I remember correctly this class is meant as a data store and then the getter is fine".*
Causes	*Developers' decision.*

A.2 - "Is" returns more than a Boolean

Method name is a predicate, whereas the return type is not Boolean but a more complex type allowing a wider range of values.

Internal developers

Perception	Developers resolved the inconsistency of method `isLeft` returning `float`, by removing the method (because the method was replaced by a different one) after the forgotten call to `isLeft` was replaced with the new method. The developer explained that the method was reused from elsewhere and the name was not adapted after the functionality changed.
Causes	*Evolution , reuse.*

A.3 - "Set" method returns

A set method having a return type different than `void` and not documenting the return type/values with an appropriate comment.

Internal developers

Perception A developer commented: *"Sometimes it is convenient that a 'set' method returns the old or the new value"*. However, two of the LA instances that internal developers resolved after we pointed out the inconsistency were of type A.3. One occurred in method `setConnectionAsSharingClient` returning `Map`; the LA was resolved by improving the (*Javadoc*) documentation, explaining the return type and values. The other instance occurred in method `setAnimationView`, returning `AnimationView`. The changes applied to resolve it impacted 3 files (see Figure 8.1). The inconsistency was resolved by: i) improving the *Javadoc* explaining that the old value of the attribute is returned (class `NotificationManager`) ii) renaming the local variable `result` to `oldView` in the child class to reflect that the result contains the old value (class `NotificationManagerImpl`), and iii) renaming the attribute `myAnimationView` to `myOriginalAnimationView` in class `DialogIml` which contains the result of `setAnimationView`, to reflect that it contains the old value.

Causes *Evolution.*

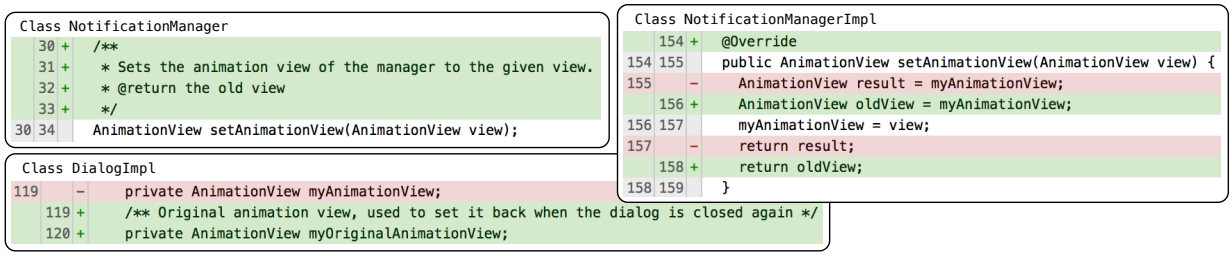


Figure 8.1 Changes applied to resolve an occurrence of A.3—`setAnimationView`.

A.4 - Expecting but not getting a single instance

Method name indicates that a single object is returned but the return type is a collection.

Internal developers

Perception Developers expressed the need to rename method `getMeetingMember` returning `List<MeetingMemberDTO>` and `getAppointmentByRange` returning `List<Appointment>`; and to comment method `getServersOption` returning `ListOption<WebDavServerDescriptor>`.

Causes *Evolution.*

B.1 - Not implemented condition

The method' comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.

Internal developers

Perception The example we pointed out is documented as: *"Release the current detector and load new detector from file (if detector_file_name is not 0). Return true on success."*, whereas its implementation always returns *false*. The developer shared that *"the code is part of a legacy module and it will be removed with the next major library update"*.

B.2 - Validation method does not confirm

A validation method that neither provides a return value informing whether the validation was successful, nor it documents how to proceed to understand.

Internal developers

Perception Method `validateSnaps` with return type `void`, is an example of B.2, that was renamed to `processSnaps` after we pointed out the inconsistency. Other examples of this LAs where developers expressed a need for renaming are methods `checkVertices` and `checkCurrentState`. The 2 examples where developers decided not to take an action are method `validateActivatedProfiles` which in case of invalid profile notifies the user with a warning; method `checkRecordingFile` where the developer commented *"a method that starts with the name "check" has a special validation meaning is new to me."*

Causes *Evolution, not enough thought.*

B.3 - "Get" method does not return

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 is stored and how to obtain it.

Internal developers

Perception For the two examples of this LA that we showed to the internal developers, they would undertake a renaming. For method `getTaskAttributes` the developer suggested to rename the parameter `id2value` making it clear that it will hold the result. For method `getUpstramProjects` a developer commented: *"Some might say that this is OK as it's a helper method for recursion when building the tree. I wouldn't".*

B.4 - Not answered question

The method name is in the form of predicate, whereas nothing is returned.

Internal developers

Perception After we pointed out method `isSnapped`, the code was removed as it was not used anymore. Another developer suggested to rename method `isLastWindow`.

Causes *Not enough thought.*

B.5 - Transform method does not return

The method name suggests the transformation of an object, however there is no return value and it is not clear from the documentation where the result is stored.

Internal developers

Perception The example we showed to a developer is method `PMCamera_Global3dToLocal3d` with `void` return type. The developer decided not to undertake an action *"to save resources—instead of creating a new object and return it, it is convenient to store the result in a parameter."*

Causes *Developers' decision.*

B.6 - Expecting but not getting a collection

The method name suggests that a collection should be returned, but a single object or nothing is returned.

Internal developers

Perception Examples of this LA where developers would undertake a renaming are method `getRows` returning `int` where developer suggested `getHeight` as more appropriate name; method `getStates` returning `State`. Examples where developers consider the practice acceptable are method `getBounds` returning `Dimension`; method `getValues` returning `bool` where the result is stored in parameter `values` and the returned value *"indicates success or failure"*.

Causes *Evolution*, *Not enough thought.*

C.1 - Method name and return type are opposite

The intent of the method suggested by its name is in contradiction with what it returns.

Internal developers

Perception Regarding method `exit_transport_impl` returning `Enter_Transport`, internal developers would rename it, however, they were not certain about the new name *"in English there isn't a word (that I know of) which bundles together 'enter' and 'exit"'*. Another example of this LA is method `player_enemy_impl` returning a `Player_Ally`, where one of the developers justified the decision as part of the design. However, other developers of the project would rename the return type to reflect both states.

Causes *Developers' decision.*

D.1 - Says one but contains many

An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.

Internal developers

Perception Internal developers suggested renaming for attribute `mInstalledPackageInfo` of type `PackageInfo[]`. Regarding attribute `projectorImage` of type `IplImage[]` a developer shared *"Could go either way - change or keep. Maybe rename to projectorImagePyramid (because it is one image at different resolutions) but it gets too long."*. One developer expressed a concern regarding the LA as follows: *"There are technical terms that will most likely sound like plural to an expert of the domain"*.

D.2 - Name suggests Boolean but type does not

An attribute name suggests that its value is *true* or *false*, while its declaring type is not Boolean and the declared type and values are not documented.

Internal developers

Perception	One questionnaire containing an example of this LA was answered. For attribute `_depends` of type `String`, the developer says that the name is well chosen as it matches standards of an imported library. The same developer also find it obvious that the field contains a reference to the packages on which the class depends.
Causes	*Developers' decision.*

E.1 - Says many but contains one

Attribute name suggests multiple objects, but its type suggests a single one.

Internal developers

Perception	Developers would resolve this LA by changing the implementation for attribute `flags` of type `unsigned char` containing multiple bit flags by *"expanding it to become a bitfield"*. An example where developers, perceived the *"inconsistency too minor to introduce changes to code working for years"* is attribute named `codecs` of type `ImageCodecInitializer` which is an initializer for multiple codecs.
Causes	*Not enough thought*, *reuse*, *developers' decision.*

F.1 - Attribute name and type are opposite

The name of an attribute is in contradiction with its type as they contain antonyms.

Internal developers

Perception	Attribute `top_index` of type `bottom_index` is an example of this LA that internal developers would rename.

F.2 - Attribute signature and comment are opposite	
Attribute declaration is in contradiction with its documentation.	
Internal developers	
Perception	Developers believe that no action need to be undertaken for `GC_start_time` documented as *"Time at which we stopped world."* because *"stop (the world) is probably synonym to start for GC people"*. An example where developers believe that a renaming is needed is `isOrdered` commented as *True if the underlying table is BTREE_UNORDERED.*

8.3 Discussions

The perception of internal developers is generally consistent with the one of external developers, i.e., the majority of the internal developers consider LAs as poor/very poor practices. However, the proportion is lower (51 vs. 69%), because sometimes the participants felt that, based on the context, a particular practice was acceptable. Also, in 56% of the cases participants suggested the LAs should be removed because they can affect program comprehension. We also found evidence of cases where those LAs were resolved in recent releases of the projects.

8.3.1 Threats to Validity

Conclusion validity: Due to the limited number of data points, we did not perform any statistical test—we discussed results qualitatively rather than quantitatively. Thus, threats to conclusion validity are not applicable in this study.

Construct validity: As for the study with external developers, we manually validated the occurrences of the LAs that we showed to participants and we selected a sample representative of different kinds of LAs. To measure participants' perception, we again used a Likert scale (Oppenheim, 1992).

Internal validity: A threat for the study is that *internal* developers could have been more lenient when judging their own code. We mitigated this threat by asking them to motivate their answer. Overall, we found a pretty high proportion of poor/very poor perception of LAs. Finally, our results may have been impacted by the fact that participants in *Study II* only validated a subset of the LAs. More data points for each LA may produce different results.

External validity: In terms of objects, the study has been conducted on eight projects. As for the study with external developers we selected projects that are very different in terms of

size and application domain. In addition, for this study we selected open-source and closed-source projects written in Java and C++. In terms of subjects, the study involved both professionals (from industry and from the open-source community)—developers of projects from which the LAs were detected.

8.4 Conclusion

In Chapter 7, we showed that 69% of the *external* developers evaluated code containing LAs as poor or very poor practices. In this chapter, we investigated the perception of *internal* developers. We asked 14 (*internal*) developers of 7 open-source Java and 1 C++ commercial projects to provide us their perception of LAs that we found in the code of their projects. Also in this case, the majority of developers (51%) evaluated LAs as poor or very poor practices. The percentage is lower than the one observed in the study with *external* developers, because in some cases *internal* developers explained that the particular context and circumstance made the detected LAs as an acceptable practice. When asked why the LAs were possibly introduced—and developers had elements to answer—they pointed out maintenance activities—e.g., done by developers different from the original code authors—that deteriorated the lexicon quality or lack of attention to naming conventions/comments. For a conspicuous proportion of LAs (56%) developers highlighted that such LAs should possibly be removed.

Interestingly, in 10% of the cases developers had already removed the inconsistencies that we pointed out. From developers' comments, we deduce that it might be more useful to point out LAs as the developer writes source code—e.g., using our LAPD Checkstyle plugin—so that the improvement can be done on-the-fly. It seems that whether on not developers remove LAs also depends on the impact that this can have on the whole project. In other words, developers may be less prone to improve the lexicon if this has a large impact on the code, as such change can be too risky.

CHAPTER 9

FACTORS IMPACTING THE IMPROVEMENT OF THE LEXICON

Highlight: In Chapters 7 and 8, we showed that both *external* and *internal* developers perceive code that contain LAs as poor or very poor. However, *internal* developers' explanations and the large proportion of yet unresolved LAs suggest that there may be other factors that impact the decision of removing LAs, which is often done through renaming. Thus, in this chapter, our objective is to understand developers habits regarding the evolution of the source code lexicon and in particular to investigate factors that may prevent developers from improving the lexicon—i.e., from renaming.

When the source code of a program evolves (Lehman, 1980), its identifiers evolve too (Abebe *et al.*, 2009a). Thus, identifier renaming, i.e., the activity during which an entity—e.g., a local variable, a method, or a class—changes its name, has a paramount importance in software evolution [1].

We expect that developers rename when they feel that the name of an entity is not (anymore) consistent with its functionality or when such a name may easily create comprehension problems. In fact, many of the identifiers tagged as having 'poor' quality—e.g., LAs (Chapter 6.1) and previous works (Deißenböck et Pizka, 2005; Abebe *et al.*, 2009b; Lawrie *et al.*, 2007a)—can be improved, i.e., resolved, by renaming. However, Antoniol *et al.* (2007) show that the evolution of source code lexicon is more limited compared to the evolution of the structure of the source code. They argue that the limited ability to evolve the source code lexicon is partially due to the cost of building a mental model of the system through its lexicon. They also suggest that other factors for the limited evolution may be due to the lack of advanced tool support for lexicon construction, documentation, and evolution. In *Study II* (Chapter 8), we observe that *internal* developers solved only 10% of the examples containing LAs, leading us to believe that, indeed, other factors may play a role in developers' decision whether an entity will be renamed or not.

In this chapter, we are interested in developers' habits regarding identifier renaming. In particular, we want to understand when do developers rename, whether they consider renaming as a straightforward activity, what factors would prevent them to rename, and what

1. Renaming is *per se* considered a refactoring activity (Fowler, 1999). In this chapter, we focus only on how developers change the source code lexicon rather than on how the source code is restructured.

are the renamings that they would benefit from if suggested by automatic recommendation systems.

9.1 Survey Design

The *goal* of the survey is to understand developers' habits regarding identifier renaming in the context of OO programming regardless whether the renaming is performed to remove an LA or not. The survey is designed from a researcher *perspective* with the *purpose* of gaining insights about possible problems that prevent lexicon to evolve naturally.

Specifically, the survey aims at answering the following research questions:

RQ1: *How Often do Developers Rename?*

RQ2: *Is Renaming Straightforward?*

RQ3: *What Factors would Prevent Developers to Rename?*

RQ4: *What Types of Renamings are Useful to be Automatically Recommended?*

In the following, we report details of how the survey has been planned and conducted.

9.1.1 Participants

We invited 739 developers, via e-mail using a convenience sampling (Groves *et al.*, 2009), involving developers from the industry and open-source communities. Although we profile survey participants based on their background, their identity is kept anonymous for confidentiality purposes. 71 developers responded to the survey resulting in a response rate close to 10% as expected (Groves *et al.*, 2009).

9.1.2 Survey Procedure

The survey was designed as an online questionnaire. First, we clarified the vocabulary we use and the context of the survey, i.e., we told participants that *"Identifier renaming consists of changing the name of an entity, where an entity, in the context of Object Oriented programming, is a class, interface, attribute, method, constructor, parameter, or local variable. By recommending identifier names or identifier renaming we mean suggesting a better name from the beginning (at the time of naming) or at the time of renaming."* Next, we ask specific questions allowing us to answer our research questions. We detail the questions in Appendix C. At any point, participants were free to decide not to answer a question by selecting the option 'No opinion' or interrupt the study.

9.2 Survey Results

This section reports the results of the survey and provides both quantitative 9.2.1 and qualitative 9.2.2 analyses.

9.2.1 Quantitative Analysis

First, we report information about participants' background. In particular, Figure 9.1 shows statistics regarding the native language of the participants, whereas Figure 9.2 reports their years of experience in industrial and open-source software development.

Figure 9.3 reports how often developers rename. Only 14% of participants rename rarely (up to once per month): 46% rename occasionally (a few times per month) while 18% rename frequently (a few times per week) and 21% rename very frequently (almost every day).

Figure 9.4 indicates activities during which developers rename. Note that a participant may select more than one activity, thus the sum of the percentages is above 100%. Participants rarely perform renaming as a standalone activity (17%). Often, they rename when performing other refactorings (90%), changing the functionality (89%), adding new functionality (65%), understanding code (51%), or fixing a bug (42%).

Figure 9.5 provides insights about the opinion of participants about the cost of renaming. 35% of participants consider that renaming requires time and effort (at least in most cases); 32% consider that the cost of renaming depends on the particular case; 32% consider renaming to be straightforward (at least in most cases). Note that the sum of the above is 99% due to rounding errors.

Figure 9.6 reports results on the use of tool support for renaming. The majority of the participants (72%) use automatic tool support to perform renaming. There are however participants that rename manually (20%) and participants that perform both, manual and automatic renaming (8%).

We asked participants to share the reasons for which they recall having decided not to rename an entity; results are shown in Figure 9.7. 52% of the participants recall the reason to be the potential impact on other projects. 35% recall that the renaming was too risky, i.e., it might have introduced a bug. 25% of the participants answered that the high impact

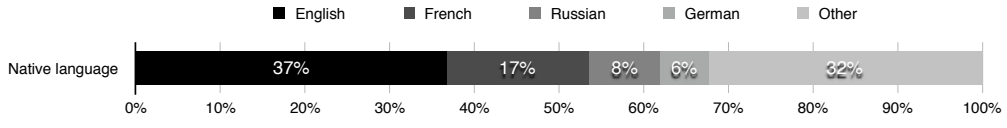


Figure 9.1 Native language of the participants.

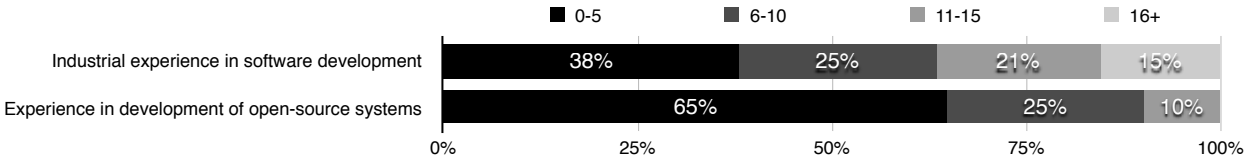


Figure 9.2 Experience of the participants in software development.

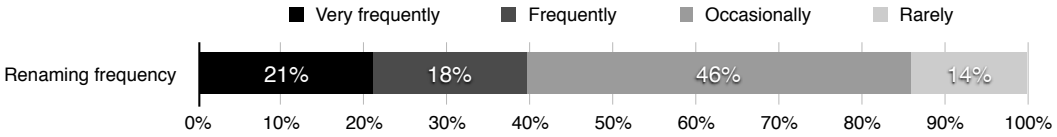


Figure 9.3 How often do developers rename?

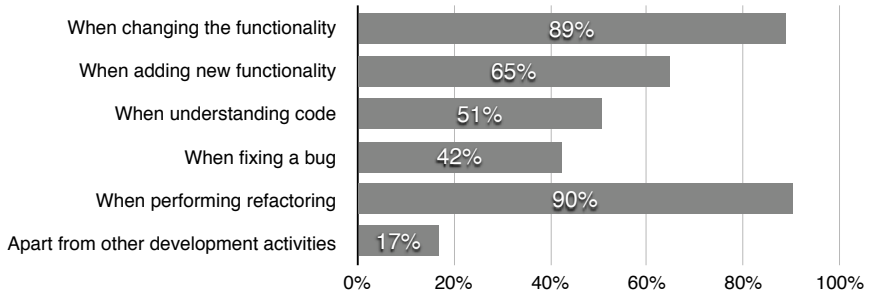


Figure 9.4 Activities accompanying renaming.

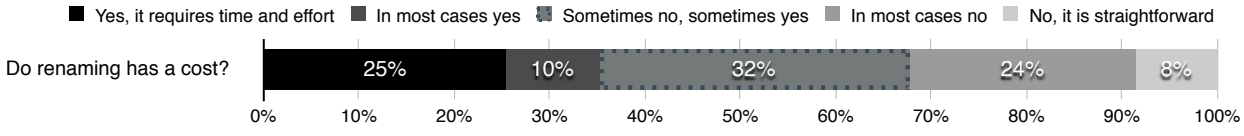


Figure 9.5 Developers' opinion on cost of renaming.

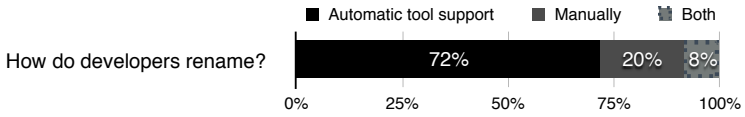


Figure 9.6 How do developers rename?

of the renaming on the project was the show-stopper and finally, 25% recall deciding not to rename because of the high effort required.

We also asked participants whether a set of predefined factors would impact the decision to undertake a renaming (Figure 9.8). The majority of participants consider important all those factors. The factor that is worth highlighting here is the impact on other projects—69% of participants say that this would definitely impact their decision.

Figure 9.9 shows when developers feel the need to rename. As expected, the majority (66%) of the participants clearly state that they will definitely rename an entity if its name is not consistent with its functionality. They made less strong statements about naming conventions, spelling errors, and hard to understand words, but still the majority of participants report that they will probably rename in such cases. Surprisingly, only 13% of participants will probably rename if an entity contains an abbreviation—the majority of participants (56%) will not rename. Finally, when the name of an entity contains a negation, e.g., `notOpen`, 30% of the participants will rename, while 46% will not.

The majority (68%) of participants see a benefit of automatic recommendations for renaming (Figure 9.10) provided that such recommendations are non-intrusive and offer reliable suggestions.

Finally, participants see a benefit of recommending the majority of renamings (Figure 9.11). However, we observe that fixing typos is the only type of renamings for which participants have strong opinion—42% of them consider that recommending this type of renamings is definitely useful.

9.2.2 Qualitative Analysis

In the following we summarize the main results of the survey and we seek for explanation of the quantitative results presented in Section 9.2.1. We illustrate results with comments from the participants and we complement the survey output with examples that we collected from online discussions of open-source projects (issue reports, mailing lists, and commit notes).

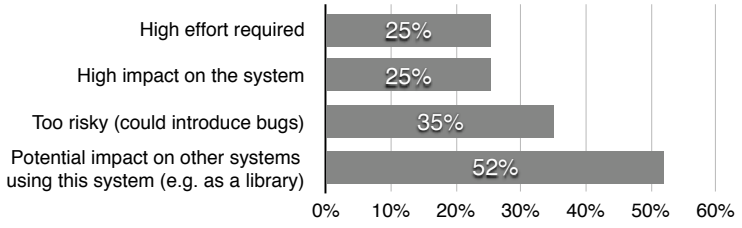


Figure 9.7 Reasons for which developers already postponed or canceled a renaming.

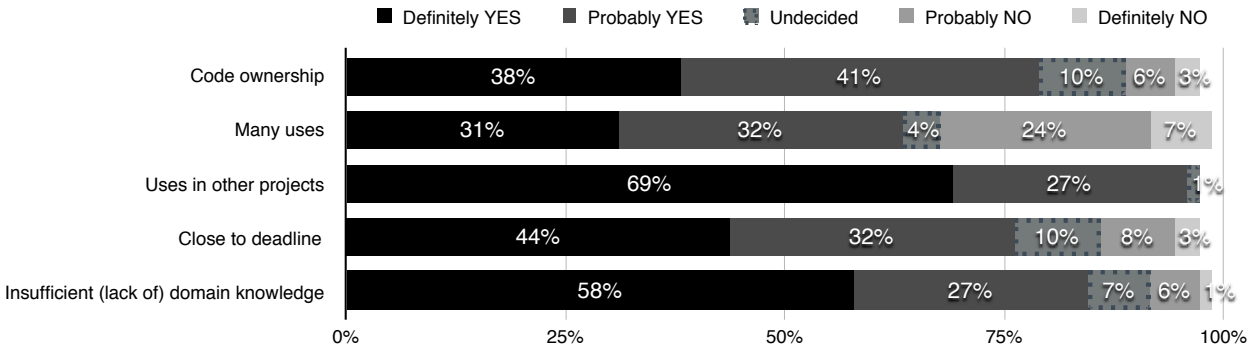


Figure 9.8 Factors impacting developers decision to undertake a renaming.

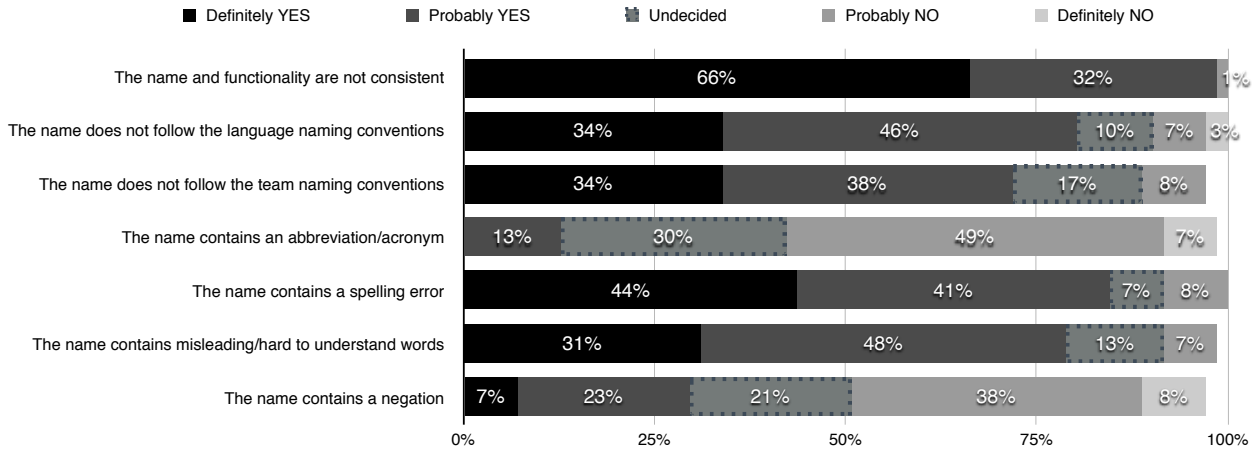


Figure 9.9 When will developers rename?

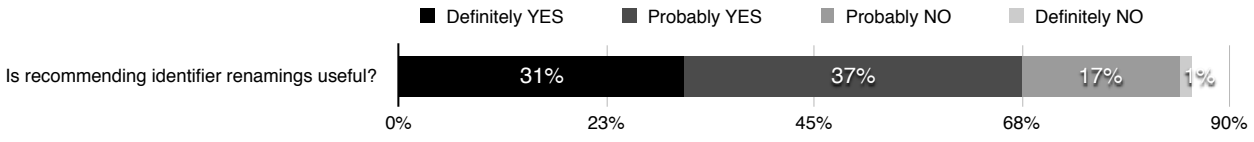


Figure 9.10 Developers' opinion on the usefulness of recommending renamings.

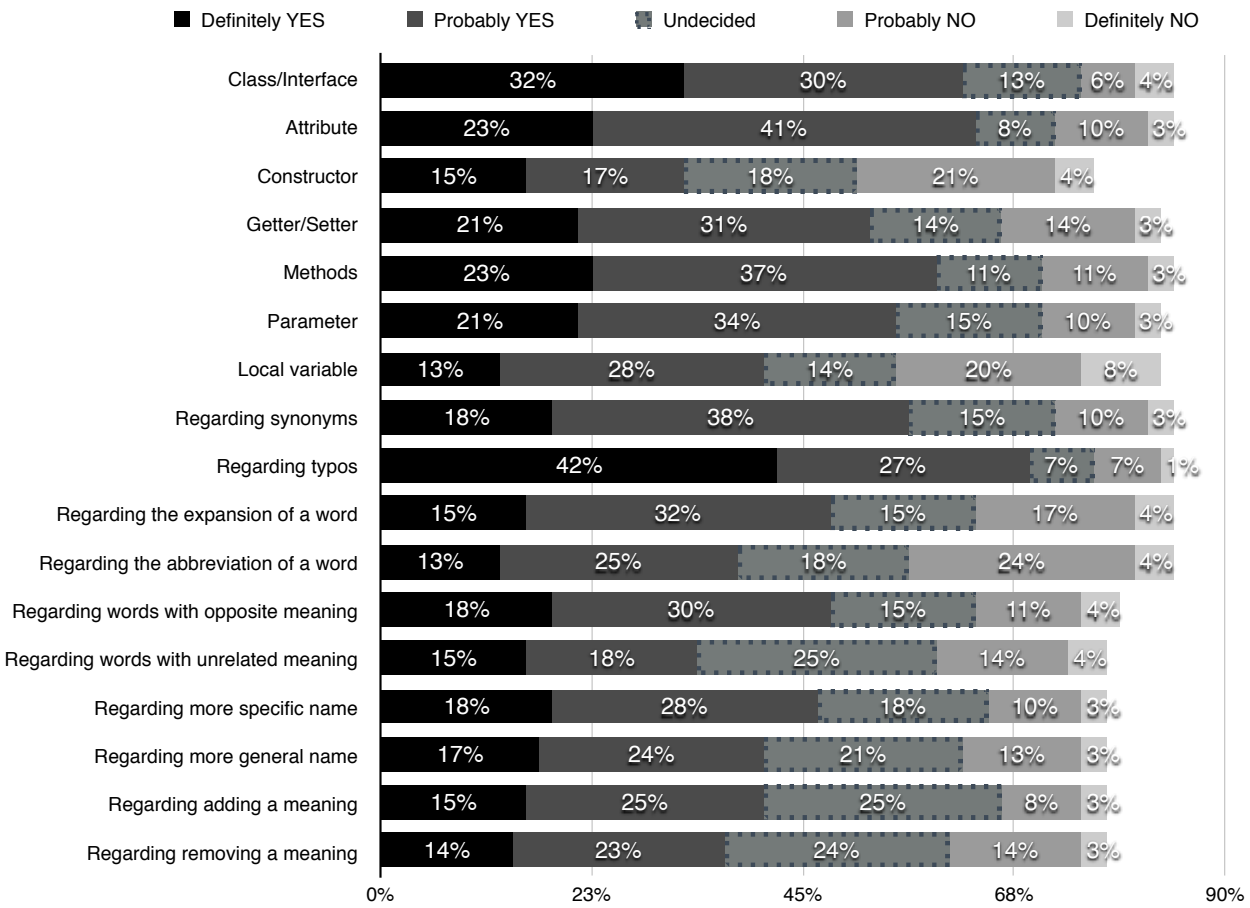


Figure 9.11 Developers' opinion on renamings that are useful to recommend.

RQ1: How Often do Developers Rename? Renaming is an activity that participants perform from almost every day (21%), a few times per week (18%), a few times per month (46%), to once per month (14%). A developer commented: *"There's a balance to be struck: - identifiers are communication, and as the code is refactored it is critical that identifiers continue to correctly describe their purpose - changing identifiers tends to break APIs, and sometimes they're used for unintended purposes, over-frequent change is not good."*

RQ2: Is Renaming Straightforward? When we asked participants whether renaming has a cost, only 8% answered that renaming is straightforward. 24% of participants think that in most cases renaming has no cost, often due to the availability of automatic tool support. Indeed the majority of participants (72%) use automatic tool support to perform renaming, although 20% rename manually and 8% use a mix of both, i.e., rename manually and automatically. 32% of participants believe that the cost of renaming depends on the particular case: *"Renaming identifiers that belong to non-local context (e.g., public or protected methods) has a potentially massive cost associated with breaking the interfaces between components. Otherwise it is typically a rather cheap and non-disruptive exercise that may have end benefit of more readable and consistent code. Another element of cost and risk is when the identifiers are being bound to at runtime only (e.g., when classes are loaded by name or methods are bound by name). It is not always easy to trace all such use cases in a large system."* Indeed, renaming an entity that is part of a public API of a program has a higher cost as it breaks backward compatibility and increases the integration cost of the program in client programs. 10% of participants believe that in most cases renaming has a cost, and finally 25% answer that renaming definitely requires time and effort. Another example where renaming has a cost is when the team uses code reviews, as developers must schedule a code review and justify their decision. A developer indicated that code reviews impact the frequency of renaming *"because you appear negatively to the boss when asking for a review on a 'too minor improvement"'*. The cost of renaming also includes the cost of finding a proper name and assuring that the new name reflects the purpose of the entity in all scenarios that it is used. Quotes like *"I have the feeling that your method name is not good [..]"* for method `getBufferForWrite` in an Eclipse issue report (issue #332248) indicates that, indeed, developers spend time understanding the rationale behind names that are chosen by other teammates.

RQ3: What Factors would Prevent Developers to Rename? It also appears that, although necessary, some renamings are delayed. After discussing the difference between

the term "delete" and "remove", an ArgoUML developer concluded that: *"[..] maybe I shall rename these after next release"* (issue #2938). We asked participants to share reasons for which they recall having decided not to rename an entity. 52% recall the reason to be the potential impact on other projects. A developer explains: *"As a middleware developer, providing a stable API is paramount for clients. There are numerous cases where we would not rename a class or method despite an obviously better name being proposed, in order to minimize the cost of integrating new versions."* 35% recall that the renaming was too risky, i.e., it might have introduced a bug—a developer recalls: *"I encountered a problem when my colleague wrote Java code which uses reflection. I avoided renaming some classes/methods which will be inspected by the reflection, since doing so can introduce unpredictable bugs."* 25% of participants answered that the high impact of the renaming on the project was the show-stopper and finally, 25% recall deciding not to rename because of the high effort required: *"I'm not touching poorly-worded APIs which are shared across multiple projects - the cost of the change does not justify it [..]."* Participants also shared that the impact on other developers is sometimes decisive: *"If too many people in the company know a thing by name X it's sometimes better to keep it even when name Y is more descriptive."* Other factors impacting the decision to undertake a renaming are insufficient domain knowledge (85% of participants), code ownership (79%), and close deadline (76%).

RQ4: What Types of Renamings are Useful to be Automatically Recommended? Although participants consider useful to recommend the majority of the renamings, only renaming towards correcting typos seems to receive strong definite support (42%). The main reason is that many of the participants are concerned with the accuracy of such automatic recommendations. This is particularly true for renamings where the meaning of the new identifier is not preserved. A developer explains that recommending renamings *"really depends on have automated design analysis sufficiently good to know that the "suggested" name is better"* and that it requires that one is able to *"correctly infer design & domain meaning and suggest an appropriate balance of conciseness & specificity in these two dimensions."* Another developer shares that recommendations tools may be beneficial as some developers will take the time to explore the alternative names but fears that others may simply *"put less thought into their names (if they feel they can rely on the tool to do the thinking for them)"* and end up *"renaming back and forth"*.

9.3 Discussions

Results show that renaming is a frequent activity that 39% of participants perform from a few times per week to almost every day. Only 8% of the participants consider renaming

to be straightforward. Some of the main factors that can prevent developers from renaming are insufficient domain knowledge (85% of the participants), code ownership (79%), close deadline (76%), the potential impact on other projects (52%), and the risk of introducing a bug (35%).

9.3.1 Threats to Validity

Conclusion validity: We do not perform any statistical test, thus threats to conclusion validity are not applicable in this study.

Internal validity: When asking participants questions, we formulate a set of predefined answers—to limit the required effort—thus possibly affecting the internal validity of the study as developers may limit their answers to the predefined list. To limit this threat, we ask participants to recall examples from their past and we provide the possibility to report customized answers.

Construct validity: To limit the threats to construct validity, questions must be carefully designed in language that is clear and understandable by the participants. To this end, we clarified the vocabulary we use throughout the survey by defining upfront what we mean by renaming and recommendation of renaming. For each question, we also provide the possibility for participants to report if they did not understand the question; thus not biasing the results of the survey.

External validity: Although we cannot really ensure full diversity (Nagappan *et al.*, 2013), the study involved developers from both the industrial and the open-source communities.

9.4 Conclusion

This chapter provides evidence that the evolution of source code lexicon is not a straightforward activity. 71 developers of industrial and open-source projects identified several factors that prevent them from renaming when they feel the need to do so. We also show the need for better tool support to facilitate developers' task—e.g., by handling renaming across projects possibly written in different programming languages. The study we performed also confirms our hypothesis that it may be more useful for developers to point out LAs at the time of writing code.

CHAPTER 10

CONCLUSION AND FUTURE DIRECTIONS

Program comprehension is a key activity during software development and maintenance as although frequently performed—even more often than actually writing code (Kersten et Murphy, 2005)—it is a difficult activity (Goldberg, 1987). The difficulty to understand a program increases with its complexity and as a result comprehension is, in the best-case scenario, more time consuming but can also lead to introducing faults in the program. Several works attempt to identify complex and fault prone programs using structural measures for program complexity. However, from early theories studying developers' behavior while understanding a program we know that identifiers and comments—i.e., the program lexicon—are part of the factors that affect the psychological complexity of a program, i.e., factors that make a program difficult to understand and maintain by humans (Weissman, 1974; Brooks, 1983). Early research also suggest that the difficulty may come from contradictions between a program' code and its comment (Brooks, 1983).

Thus, in this dissertation we formulated the following thesis:

> Our thesis is that poor lexicon negatively impacts the quality of software, that the quality of the lexicon depends on the quality of individual identifiers but also on the consistency among identifiers from different sources (name, implementation, and documentation), and that the definition of practices that result in poor quality lexicon increases developer awareness and thus contributes to the improvement of the lexicon.

To validate our thesis, we provided the following main contributions:

— We brought evidence that measures evaluating the lexicon quality are an asset for fault explanation and prediction. In particular, we showed that measures such as LBS (Abebe *et al.*, 2009b) and HEHCC (Arnaoudova *et al.*, 2010) improve the performance of explanatory and prediction models when added to structural complexity measures such as the CK metric suite and LOC, respectively. The advantage of measures evaluating the lexicon quality with respect to structural measures is that lexicon measures are easier to understand, interpret, and eventually avoid or fix.

— We also contributed to the improvement of the lexicon consistency by defining a new family of antipatterns, i.e., Linguistic Antipatterns (LAs), for program entities (i.e., methods and attributes). In particular, we defined a catalog of Linguistic An-

tipattern (LA) related to inconsistencies among the name, implementation, and documentation (i.e., comment) of a program entity. We evaluated the catalog from the point of view of industrial and open-source developers and we showed that the majority of the developers perceive LAs as poor practices and therefore must be avoided. Provided that static analysis tools struggle to gain acceptance among developers, we distilled a subset of *canonical LAs* (see Table 10.1) recognized by external (column **SI**) and–or internal (column **SII**) developers as poor practices. *Canonical LAs* are 1) perceived as 'Poor' or 'Very poor' by at least 75% of the *external* developers, 2) perceived as 'Poor' or 'Very poor' by all *internal* developers (due to the limited number of data points), or 3) those LAs for which internal developers undertook an action to resolve them. *Canonical LAs* would likely gain developer acceptance if their detection is accurate.

— We perform a survey with open-source and industrial developers to gain insights about renaming and understand why they resolved only part of the examples containing LAs. We report factors that may prevent the improvement of the source code lexicon.

Table 10.1 Canonical LAs.

		SI	**SII**
A.2	*"Is" returns more than a Boolean*		✓
A.3	*"Set" method returns*	✓	✓
B.2	*Validation method does not confirm*		✓
B.3	*"Get" method does not return*	✓	
B.4	*Not answered question*	✓	✓
B.6	*Expecting but not getting a collection*	✓	
C.2	*Method signature and comment are opposite*	✓	
D.2	*Name suggests Boolean but type does not*	✓	
E.1	*Says many but contains one*	✓	
F.1	*Attribute name and type are opposite*	✓	✓
F.2	*Attribute signature and comment are opposite*	✓	✓

10.1 Limitations

Our approach and results are subject to the following limitations:

Identifier Quality and Code Quality

We provide evidence that metrics evaluating the quality of source code lexicon capture additional information that is not captured by metrics evaluating the structure of the code. We also show that this additional information is an asset as it improves software fault explanation and prediction. However, our results do not allow us to claim any causal relation between the quality of source code lexicon and software quality.

We investigate the relation between lexicon quality (i.e., HEHCC and LBS) and fault proneness in multiple versions of three Java projects only. More projects are needed for Java as well as other programming languages before being able to generalize the results.

To approximate context coverage we use the textual similarity between entities using LSI. Although LSI is known to deal with synonymy and polysemy, a domain ontology such as WordNet (Miller, 1995) may lead to a more accurate context representation.

Limitations of LAs

We defined LAs and group them into categories based on a close inspection of source code examples from several open-source projects written in Java. Thus, they may not be representative inconsistencies of the entire population of source code entities.

To detect LAs we rely on tools—ontological databases such as WordNet (Miller, 1995) and natural language parsers such as the Stanford CoreNLP (Toutanova et Manning, 2000)—explicitly conceived to process natural language documents rather than source code.

To evaluate the precision of LAPD we manually validated a random sample of LA examples occurring in four open-source Java projects. Thus, result from a validation by the original developers and on projects written in other programming languages may produce different results.

The evaluation of LAs is threatened by the fact that developers' perceptions are bound to the particular examples rather than the practice itself. When asking participants to evaluate code snippets containing LAs, we formulate a specific question thus possibly affecting the internal validity of the study as participants may guess the expected answer (Shull *et al.*, 2007). Also, as participants in *Study I* are external to the project, the lack of domain knowledge may have impacted their perception. Participants in *Study II* are subject to the threat that they could have been more lenient with their own code. Also, due to the limited number of data points in *Study II*, we did not perform any particular analysis and we discuss results qualitatively rather than quantitatively.

10.2 Future Directions

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

Lexicon Quality and Code Quality

In the future we plan to study the relation between LAs and software faults. Moreover, future studies that concentrate on a causal relation between lexicon quality and code quality are needed. For example, to provide empirical evidence that HEHCC, LBS, and LAs indeed impact program comprehension, a controlled experiment must be designed where subjects are asked to understand part of source code of a project with poor lexicon quality and a version where the lexicon was improved. The impact would be measured in terms of the degree of understanding, the time to understand, and the effort to understand—e.g., using an eye-tracking system. In addition, controlled studies evaluating whether poor identifier quality leads to fault introduction would be very valuable to the community. Such studies, would ask participants to introduce a new feature or modify an existing one in different versions of a project, i.e., with poor quality lexicon and a version with improved lexicon.

Improve the Detection of LAs

Improvements of the detection of LAs can consider project specific and domain specific knowledge. For example, in their recent work Yang et Tan (2013) propose an approach to mine semantically related words in a project or multiple projects from the same domain. Similar work has been done by Howard *et al.* (2013) where the authors mine semantically similar words across projects from multiple domains. Also, in their recent work, Gupta *et al.* (2013) proposed an approach for POS tagging of source code identifiers and showed that the approach parses identifiers 10 to 20 percent more accurately. The LAPD would certainly benefit from such approaches and increases the precision of detecting LAs.

Benefit of Reporting Poor Lexicon Quality On-The-Fly

As our results show, developer's decision whether to improve the lexicon through renaming may sometimes be affected by several factors. We thus hypothesize that an on-the-fly detection of LA would be more beneficial. It would be interesting to empirically validate this hypothesis. In addition, it would be also beneficial to study whether systematically reporting LAs decreases the number of examples containing LAs over time. If this is indeed the case, we may hope that such on-the-fly tools—reporting LA but also other poor lexicon quality practices—can be integrated in the environment in which students learn to program and improve the quality of the programs over time.

Study the Impact of Removing LAs

From our study with developers on factors that may prevent renaming, we recommend that future work also provides support for renaming program entities across projects and particularly across projects written in different programming languages. Further support is also needed to rename entities used in reflection. Finally, some developers expressed reluctance to rename an entity because *"if too many people in the company know a thing by name X it's sometimes better to keep it even when name Y is more descriptive"*. We encourage future research to study whether indeed renaming may have such negative consequences. Such empirical studies may consider three main treatments, for instance 1) the program where the renaming is not performed, 2) the program where the renaming is performed, and 3) the program where the renaming is performed and documented—e.g., using a tool such as REPENT (Arnaoudova *et al.*, 2014).

REFERENCES

ABBES, M., KHOMH, F., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 181–190.

ABEBE, S. L., HAIDUC, S., MARCUS, A., TONELLA, P. et ANTONIOL, G. (2009a). Analyzing the evolution of the source code vocabulary. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 189–198.

ABEBE, S. L., HAIDUC, S., TONELLA, P. et MARCUS, A. (2009b). Lexicon bad smells in software. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 95–99.

ABEBE, S. L., HAIDUC, S., TONELLA, P. et MARCUS, A. (2011). The effect of lexicon bad smells on concept location in source code. *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 125–134.

ABEBE, S. L. et TONELLA, P. (2010). Natural language parsing of program element names for concept extraction. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 156–159.

ABEBE, S. L. et TONELLA, P. (2011). Towards the extraction of domain concepts from the identifiers. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 77–86.

ABEBE, S. L. et TONELLA, P. (2013). Automated identifier completion and replacement. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 263–272.

AL DALLAL, J. et BRIAND, L. C. (2012). A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21, 8:1–8:34.

ANQUETIL, N. et LETHBRIDGE, T. (1998). Extracting concepts from file names; a new file clustering criterion. *Proceedings of the International Conference on Software Engineering (ICSE)*. 84–93.

ANTONIOL, G., GUÉHÉNEUC, Y.-G., MERLO, E. et TONELLA, P. (2007). Mining the lexicon used by programmers during software evolution. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 14–23.

ARISHOLM, E., BRIAND, L. C. et FUGLERUD, M. (2007). Data mining techniques for building fault-proneness models in telecom Java software. *Proceedings of the International Symposium on Software Reliability (ISSRE)*. 215–224.

ARNAOUDOVA, V., DI PENTA, M., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2013). A new family of software anti-patterns: Linguistic anti-patterns. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 187–196.

ARNAOUDOVA, V., ESHKEVARI, L., DI PENTA, M., OLIVETO, R., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2014). REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering (TSE)*, 40, 502–532.

ARNAOUDOVA, V., ESHKEVARI, L. M., OLIVETO, R., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2010). Physical and conceptual identifier dispersion: Measures and relation to fault proneness. *Proceedings of the International Conference on Software Maintenance (ICSM) - ERA Track*. 1–5.

BAEZA-YATES, R. A. et RIBEIRO-NETO, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BASILI, V. R., BRIAND, L. C. et MELO, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering (TSE)*, 22, 751–761.

BAVOTA, G., DE LUCIA, A. et OLIVETO, R. (2011). Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software (JSS)*, 84, 397–414.

BAVOTA, G., LUCIA, A., MARCUS, A. et OLIVETO, R. (2013). Using structural and semantic measures to improve software modularization. *Empirical Software Engineering (EMSE)*, 18, 901–932.

BELL, R. M., OSTRAND, T. J. et WEYUKER, E. J. (2011). Does measuring code change improve fault prediction? *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. 2:1–2:8.

BINKLEY, D., DAVIS, M., LAWRIE, D. et MORRELL, C. (2009). To CamelCase or Under_score. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 158–167.

BINKLEY, D., FEILD, H., LAWRIE, D. et PIGHIN, M. (2007). Software fault prediction using language processing. *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*. 99–110.

BINKLEY, D., HEARN, M. et LAWRIE, D. (2011). Improving identifier informativeness using part of speech information. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 203–206.

BREIMAN, L. (2001). Random forests. *Machine Learning*, 45, 5–32.

BRIAND, L. C., DALY, J. W. et WÜST, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering (EMSE)*, 3, 65–117.

BRIAND, L. C., WÜST, J., DALY, J. W. et PORTER, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software (JSS)*, 51, 245–273.

BROOKS, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543–554.

BROWN, W. J., MALVEAU, R. C., III, H. W. M. et MOWBRAY, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc.

BUSE, R. et WEIMER, W. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering (TSE)*, 36, 546–558.

BUSE, R. P. et WEIMER, W. R. (2008). A metric for software readability. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 121–130.

BUTLER, S., WERMELINGER, M., YU, Y. et SHARP, H. (2009). Relating identifier naming flaws and code quality: An empirical study. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 31–35.

BUTLER, S., WERMELINGER, M., YU, Y. et SHARP, H. (2010). Exploring the influence of identifier names on code quality: An empirical study. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 156–165.

CAPRILE, B. et TONELLA, P. (1999). Nomen est omen: Analyzing the language of function identifiers. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 112–122.

CAPRILE, B. et TONELLA, P. (2000). Restructuring program identifier names. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 97–107.

CATALDO, M., MOCKUS, A., ROBERTS, J. et HERBSLEB, J. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering (TSE)*, 35, 864–878.

CHAUDHARY, B. D. et SAHASRABUDDHE, H. V. (1980). Meaningfulness as a factor of program complexity. *Proceedings of the ACM Annual Conference*. 457–466.

CHIDAMBER, S. R. et KEMERER, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20, 476–493.

COLLARD, M., KAGDI, H. et MALETIC, J. (2003). An XML-based lightweight C++ fact extractor. *Proceedings of the International Workshop on Program Comprehension (IWPC)*. 134–143.

COOK, T. D. et CAMPBELL, D. T. (1979). *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Wadsworth Publishing.

CORRITORE, C. L. et WIEDENBECK, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54, 1–23.

COVER, T. M. et THOMAS, J. A. (2006). *Elements of Information Theory*. Wiley Series in Telecommunications John Wiley & Sons., second edition.

D'AMBROS, M., LANZA, M. et ROBBES, R. (2010). An extensive comparison of bug prediction approaches. *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*. 31–41.

DE LUCIA, A., DI PENTA, M. et OLIVETO, R. (2011). Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering (TSE)*, 37, 205–227.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K. et HARSHMAN, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science (JASIS)*, 41, 391–407.

DEIßENBÖCK, F. et PIZKA, M. (2005). Concise and consistent naming. *Proceedings of the International Workshop on Program Comprehension (IWPC)*. 97–106.

DEIßENBÖCK, F. et PIZKA, M. (2006). Concise and consistent naming. *Software Quality Journal (SQJ)*, 14, 261–282.

DI MARTINO, S., FERRUCCI, F., GRAVINO, C. et SARRO, F. (2011). A genetic algorithm to configure support vector machines for predicting fault-prone components. *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES)*. 247–261.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K. D., GARG, V., MURPHY, G. C., NAGAPPAN, N. et AHO, A. V. (2008). Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering (TSE)*, 34, 497–515.

ELISH, K. O. et ELISH, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software (JSS)*, 81, 649–660.

ENSLEN, E., HILL, E., POLLOCK, L. et VIJAY-SHANKER, K. (2009). Mining source code to automatically split identiers for software analysis. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 71–80.

FALLERI, J.-R., HUCHARD, M., LAFOURCADE, M., NEBUT, C., PRINCE, V. et DAO, M. (2010). Automatic extraction of a wordnet-like identifier network from software. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 4–13.

FENTON, N. E. et PFLEEGER, S. L. (1996). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, second edition.

FLURI, B., WÜRSCH, M. et GALL, H. C. (2007). Do code and comments co-evolve? on the relation between source code and comment changes. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 70–79.

FOWLER, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

GAO, J., ZHOU, M., NIE, J.-Y., HE, H. et CHEN, W. (2002). Resolving query translation ambiguity using a decaying co-occurrence model and syntactic dependence relations. *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR)*. 183–190.

GLASER, B. G. (1992). *Basics of grounded theory analysis*. Sociology Press.

GOLDBERG, A. (1987). Programmer as reader. *IEEE Software*, 4, 62–70.

GRISSOM, R. J. et KIM, J. J. (2005). *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, second edition.

GROVES, R. M., FOWLER JR., F. J., COUPER, M. P., LEPKOWSKI, J. M., SINGER, E. et TOURANGEAU, R. (2009). *Survey Methodology*. Wiley, second edition.

GUÉHÉNEUC, Y.-G., SAHRAOUI, H. et ZAIDI, F. (2004). Fingerprinting design patterns. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 172–181.

GUERROUJ, L., DI PENTA, M., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2013). TIDIER: an identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, 25, 575–599.

GUPTA, S., MALIK, S., POLLOCK, L. et VIJAY-SHANKER, K. (2013). Part-of-speech tagging of program identifiers for improved text-based software engineering tool. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 3–12.

GYIMÓTHY, T., FERENC, R. et SIKET, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE)*, 31, 897–910.

HAIDUC, S. et MARCUS, A. (2008). On the use of domain terms in source code. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 113–122.

HASSAN, A. E. (2009). Predicting faults using the complexity of code changes. *Proceedings of the International Conference on Software Engineering (ICSE)*. 78–88.

HINDLE, A., ERNST, N. A., GODFREY, M. W. et MYLOPOULOS, J. (2011). Automated topic naming to support cross-project analysis of software maintenance activities. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 163–172.

HINTZE, J. L. et NELSON, R. D. (1998). Violin plots: A box plot-density trace synergism. *The American Statistician*, 52, 181–184.

HøST, E. W. et ØSTVOLD, B. M. (2009). Debugging method names. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 294–317.

HOVEMEYER, D. et PUGH, W. (2004). Finding bugs is easy. *SIGPLAN Not.*, 39, 92–106.

HOWARD, M. J., GUPTA, S., POLLOCK, L. et VIJAY-SHANKER, K. (2013). Automatically mining software-based, semantically-similar words from comment-code mappings. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 377–386.

IBRAHIM, W. M., BETTENBURG, N., ADAMS, B. et HASSAN, A. E. (2012). On the relationship between comment update practices and software bugs. *Journal of Systems and Software (JSS)*, 85, 2293–2304.

KAMEI, Y., MATSUMOTO, S., MONDEN, A., ICHI MATSUMOTO, K., ADAMS, B. et HASSAN, A. E. (2010). Revisiting common bug prediction findings using effort-aware models. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 1–10.

KARLSEN, E. K., HøST, E. W. et ØSTVOLD, B. M. (2012). Finding and fixing java naming bugs with the Lancelot Eclipse plugin. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 35–38.

KERSTEN, M. et MURPHY, G. C. (2005). Mylar: A degree-of-interest model for ides. *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. 159–168.

KHOMH, F., DI PENTA, M. et GUÉHÉNEUC, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 75–84.

KHOMH, F., DI PENTA, M., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering (EMSE)*, 17, 243–275.

KIM, S., ZIMMERMANN, T., WHITEHEAD JR., E. J. et ZELLER, A. (2007). Predicting faults from cached history. *Proceedings of the International Conference on Software Engineering (ICSE)*. 489–498.

KPODJEDO, S., RICCA, F., GALINIER, P., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2011). Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering (EMSE)*, 16, 141–175.

LAWRIE, D. et BINKLEY, D. (2011). Expanding identifiers to normalize source code vocabulary. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 113–122.

LAWRIE, D., BINKLEY, D. et MORRELL, C. (2010). Normalizing source code vocabulary. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 3–12.

LAWRIE, D., FEILD, H. et BINKLEY, D. (2006a). Syntactic identifier conciseness and consistency. *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*. 139–148.

LAWRIE, D., FEILD, H. et BINKLEY, D. (2007a). Quantify identifier quality: An analysis of trends. *Empirical Software Engineering (EMSE)*, 12, 359–388.

LAWRIE, D., MORRELL, C., FEILD, H. et BINKLEY, D. (2006b). What's in a name? A study of identifiers. *Proceedings of International Conference on Program Comprehension (ICPC)*. 3–12.

LAWRIE, D., MORRELL, C., FEILD, H. et BINKLEY, D. (2007b). Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3, 303–318.

LAWRIE, D. J., FEILD, H. et BINKLEY, D. (2006c). Leveraged quality assessment using information retrieval techniques. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 149–158.

LEHMAN, M. M. (1980). Programs life cycles and laws of software evolution. *Proceedings of the IEEE*, 68, 1060–1076.

LIENTZ, B. P., SWANSON, E. B. et TOMPKINS, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM (CACM)*, 21, 466–471.

LIU, Y., POSHYVANYK, D., FERENC, R., GYIMÓTHY, T. et CHRISOCHOIDES, N. (2009). Modeling class cohesion as mixtures of latent topics. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 233–242.

LORENZ, M. et KIDD, J. (1994). *Object-Oriented Software Metrics*. Prentice-Hall.

MADANI, N., GUERROUJ, L., DI PENTA, M., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2010). Recognizing words from source code identifiers using speech recognition techniques. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 68–77.

MÄNTYLÄ, M. V. et LASSENIUS, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering (EMSE)*, 11, 395–431.

MARCUS, A., POSHYVANYK, D. et FERENC, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering (TSE)*, 34, 287–30.

MARCUS, M. P., MARCINKIEWICZ, M. A. et SANTORINI, B. (1993). Building a large annotated corpus of English: the penn treebank. *Journal of Computational Linguistics - Special issue on using large corpora*, 19, 313–330.

MATTHEWS, B. (1975). Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405, 442–451.

MCCABE, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, SE-2, 308–320.

MENDE, T. et KOSCHKE, R. (2009). Revisiting the evaluation of defect prediction models. *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. 7:1–7:10.

MERLO, E., MCADAM, I. et DE MORI, R. (2003). Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance: Research and Practice*, 15, 205–244.

MILLER, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM*, 38, 39–41.

MOHA, N., GUÉHÉNEUC, Y.-G., DUCHIEN, L. et LE MEUR, A.-F. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36, 20–36.

MORENO, L., APONTE, J., SRIDHARA, G., MARCUS, A., POLLOCK, L. et VIJAY-SHANKER, K. (2013). Automatic generation of natural language summaries for java classes. *Proceedings of the International Conference on Program Comprehension (ICPC)*. 23–32.

NAGAPPAN, M., ZIMMERMANN, T. et BIRD, C. (2013). Diversity in software engineering research. *Proceedings of the Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE). 466–476.

NAGAPPAN, N. et BALL, T. (2005). Use of relative code churn measures to predict system defect density. *Proceedings of the International Conference on Software Engineering (ICSE)*. 284–292.

OPPENHEIM, A. N. (1992). *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A. et POSHYVANYK, D. (2013). Detecting bad smells in source code using change history information. *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 268–278.

PALOMBA, F., BAVOTA, G., PENTA, M. D., OLIVETO, R. et LUCIA, A. D. (2014). Do they really smell bad? a study on developers' perception of code bad smells. *International Conference on Software Maintenance and Evolution (ICSME)*. To appear.

POLLOCK, L., VIJAY-SHANKER, K., SHEPHERD, D., HILL, E., FRY, Z. P. et MALOOR, K. (2007). Introducing natural language program analysis. *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 15–16.

POSHYVANYK, D. et MARCUS, A. (2006). The conceptual coupling metrics for object-oriented systems. *Proceedings of the International Conference on Software Maintenance (ICSM)*. 469–478.

PRADEL, M. et GROSS, T. (2013). Name-based analysis of equally typed method arguments. *IEEE Transactions on Software Engineering (TSE)*, 39, 1127–1143.

RAŢIU, D., DUCASSE, S., GIRBA, T. et MARINESCU, R. (2004). Using history information to improve design flaws detection. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 223–232.

RELF, P. (2005). Tool assisted identifier naming for improved software readability: an empirical study. *International Symposium on Empirical Software Engineering (ISESE)*.

ROBILLARD, M. P., COELHO, W. et MURPHY, G. C. (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering (TSE)*, 30, 889–903.

SHESKIN, D. J. (2007). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & All, fourth edition.

SHIHAB, E., JIANG, Z. M., IBRAHIM, W. M., ADAMS, B. et HASSAN, A. E. (2010). Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 4:1–4:10.

SHNEIDERMAN, B. et MAYER, R. (1975). Towards a cognitive model of progammer behavior. Technical report 37, Indiana University.

SHULL, F., SINGER, J. et SJøBERG, D. I. (2007). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

SOLOWAY, E., LAMPERT, R., LETOVSKY, S., LITTMAN, D. et PINTO, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM (CACM)*, 31, 1259–1267.

SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L. et VIJAY-SHANKER, K. (2010). Towards automatically generating summary comments for Java methods. *Proceedings of the International conference on Automated Software Engineering (ASE)*. 43–52.

SRIDHARA, G., POLLOCK, L. et VIJAY-SHANKER, K. (2011). Automatically detecting and describing high level actions within methods. *Proceedings of the International Conference on Software Engineering (ICSE)*. 101–110.

STANDISH, T. A. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering (TSE)*, 10, 494–497.

STRAUSS, A. L. (1987). *Qualitative analysis for social scientists*. Cambridge Univsersity Press.

TAKANG, A. A., GRUBB, P. A. et MACREDIE, R. D. (1996). The effects of comments and identifier names on program comprehensibility: An experiential study. *Journal of Program Languages*, 4, 143–167.

TAN, L., YUAN, D., KRISHNA, G. et ZHOU, Y. (2007). /*iComment: Bugs or bad comments?*/. *ACM SIGOPS Operating Systems Review*, 41, 145–158.

TAN, L., ZHOU, Y. et PADIOLEAU, Y. (2011). acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. *Proceedings of the International Conference on Software Engineering (ICSE)*. 11–20.

TAN, S. H., MARINOV, D., TAN, L. et LEAVENS, G. T. (2012). @tComment: Testing Javadoc comments to detect comment-code inconsistencies. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 260–269.

THUMMALAPENTA, S., CERULO, L., AVERSANO, L. et DI PENTA, M. (2010). An empirical study on the maintenance of source code clones. *Empirical Software Engineering (EMSE)*, 15, 1–34.

TIARKS, R. (2011). What maintenance programmers really do: An observational study. *Proceedings of the Workshop Software Reengineering (WSR)*. 36–37.

TOUTANOVA, K. et MANNING, C. D. (2000). Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*. 63–70.

VON MAYRHAUSER, A., VANS, A. M. et HOWE, A. E. (1997). Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9, 299–327.

WANG, X., POLLOCK, L. et VIJAY-SHANKER, K. (2014). Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of Software: Evolution and Process (JSEP)*, 26, 27–49.

WEISSMAN, L. M. (1974). *A Methodology for Studying the Psychological Complexity of Computer Programs.* Ph.D. thesis, University of Toronto.

WELKER, K. D., OMAN, P. W. et ATKINSON, G. G. (1997). Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9, 127–159.

WEYUKER, E. J., OSTRAND, T. J. et BELL, R. M. (2010). Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering (EMSE)*, 15, 277–295.

WOHLIN, C., RUNESON, P., M., H., OHLSSON, M., REGNELL, B. et WESSLÉN, A. (2000). *Experimentation in Software Engineering - An Introduction.* Kluwer Academic Publishers.

YAMASHITA, A. et MOONEN, L. (2013). Do developers care about code smells? - an exploratory survey. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 242–251.

YANG, J. et TAN, L. (2013). SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering (EMSE)*, 1–31.

YIN, R. K. (1994). *Case Study Research: Design and Methods.* Sage Publications, second edition.

ZHANG, F., MOCKUS, A., ZOU, Y., KHOMH, F. et HASSAN, A. (2013). How does context affect the distribution of software maintainability metrics? *Proceedings of the International Conference on Software Maintenance (ICSM)*. 350–359.

ZHONG, H., ZHANG, L., XIE, T. et MEI, H. (2011). Inferring specifications for resources from natural language api documentation. *Automated Software Engineering*, 18, 227–261.

ZHOU, Y. et LEUNG, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering (TSE)*, 32, 771–789.

ZIMMERMANN, T., PREMRAJ, R. et ZELLER, A. (2007). Predicting defects for Eclipse. *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*. 9:1–9:7.

ANNEXE A

LAs: Detection Algorithms

Detection algorithms

A.1—"Get" - more than an accessor: Find accessor methods by identifying methods whose name starts with 'get' and ends with a substring that corresponds to an attribute in the same class and where the attribute's declared type and the accessor's return type are the same. Then, identify those accessors that are performing more actions than returning the corresponding attribute. Cases where the attribute is set before it is returned (i.e., Proxy and Singleton design patterns) should not be considered as part of this LA. For a detection built on top of an Abstract Syntax Tree (AST) expressions other than a return statement—where the attribute is returned—can be allowed only if they are child of a conditional check for null value. Other measures for complexity, such as LOC or McCabe's Cyclomatic Complexity, can be used for a simpler but less accurate detection.
A.2—"Is" returns more than a Boolean: Find methods starting with "is" and returning a type (i.e., the return type is not `void`) that is not Boolean.
A.3—"Set" method returns: Find modifier methods (or more generally methods whose name starts with "set") and whose return type is different from `void`.
A.4—Expecting but not getting a single instance: Find methods returning a collection (e.g., array, list, vector, etc.) but whose name ends with a singular noun and does not contain a word implying a collection (e.g., array, list, vector, etc.).
B.1—Not implemented condition: Find methods with at least one conditional sentence in comments but with no conditional statements in the implementation (e.g., no control structures or ternary operators).
B.2—Validation method does not confirm: Find validation methods (e.g., method names starting with "validate", "check", "ensure") whose return type is `void` and that do not throw an exception.
B.3—"Get" method does not return: Find methods where the name suggests a return value (e.g., names starting with "get", "return") but where the return type is `void`.
B.4—Not answered question: Find methods whose name is in the form of predicate (e.g., starts with "is", "has") and whose return type is `void`.
B.5—Transform method does not return: Find methods whose name suggests a transformation of an object, (e.g., `toSomething`, `source2target`) but its return type is `void`.
B.6—Expecting but not getting a collection: The method name suggests that it returns (e.g., starts with "get", "return") multiple objects (e.g., ends with a plural noun), however the return type is not a collection.
C.1—Method name and return type are opposite: Find methods where the name and return type contain antonyms.

C.2—Method signature and comment are opposite: Find methods whose name or return type have an antonym relation with its comment.
D.1—Says one but contains many: Find attributes having a name ending with a singular noun and having a collection as declaring type.
D.2—Name suggests Boolean but type does not: Find attributes whose name is structured as a predicate, i.e., starting with a verb in third person (e.g., "is", "has") or ending with a verb in gerund/present participle, but whose declaring type is not Boolean.
E.1—Says many but contains one: Find attributes having a name ending with a plural noun, however their type is not a collection neither it contains a plural noun.
F.1—Attribute name and type are opposite: Find attributes whose name and declaring type contain antonyms.
F.2—Attribute signature and comment are opposite: Find attributes whose name or declaring type have an antonym relation with its comment.

ANNEXE B

Studied Projects

In the following we first provide the list of all studied projects in this dissertation. We then provide the particular versions of the projects used for each study.

Projects overview

ArgoUML [1]: An open-source UML modeling tool.
Apache Maven [2]: An open-source software project management tool.
Apache OpenMeetings [3]: An open-source project for meeting management.
boost [4]: An open-source set of libraries for C++.
BWAPI [5]: An open-source API for the StarCraft Brood War game.
Cocoon [6]: A Spring-based framework to build Web applications.
CommitMonitor [7]: An open-source application for monitoring repositories.
Eclipse [8]: A well known framework and Integrated Development Environment (IDE).
GanttProject [9]: An open-source project management tool.
MagicPlan [10]: A closed-source mobile application for floor plan creation.
OpenCV [11]: An open-source library for real-time image processing.
Rhino [12]: An open-source implementation of JavaScript.

Projects' versions

All tables report information about the studied versions and the LOC [13].

Table B.1 reports the projects analyzed to study the relation between *numHEHCC* and LOC for fault explanation (Chapter 4). All projects are written in Java.

Table B.2 lists the projects that we use to study whether LBS help to improve fault prediction (Chapter 5). All projects are written in Java.

1. `http://argouml.tigris.org`
2. `http://maven.apache.org/`
3. `http://openmeetings.apache.org/`
4. `http://www.boost.org/`
5. `https://code.google.com/p/bwapi/`
6. `http://cocoon.apache.org`
7. `https://code.google.com/p/commitmonitor/`
8. `http://www.eclipse.org`
9. `http://www.ganttproject.biz/`
10. `http://www.sensopia.com/english/index.html`
11. `http://opencv.org/`
12. `www.mozilla.org/rhino`
13. LOC for all projects is calculated using CLOC: `http://cloc.sourceforge.net/`.

Table B.1 Projects analyzed to study the *numHEHCC* for fault explanation.

Project	Version	Size (LOC)
ArgoUML	0.16	105K
Rhino	1.4R3	18K

Table B.3 lists the projects that we used to study LAs (Chapters 6, 7, and 8). Projects are written in Java and–or C++. For projects where we did not provide a version, we used version control (accessed on 31/05/2013).

Table B.2 Projects analyzed to study *LBS* for fault prediction.

Project	Version	Size (LOC)	Classes	
			Total	Defective
ArgoUML	0.10.1	82K	863	49
	0.12	91K	946	47
	0.14	107K	1227	93
	0.16	105K	1185	152
	0.18.1	118K	1249	52
	0.20	165K	1333	127
Eclipse	1.0	475K	4596	96
	2.0	792K	5985	163
	2.1.1	991K	6748	98
	2.1.2	992K	6750	78
	2.1.3	993K	6754	149
Rhino	1.4R3	18K	94	66
	1.5R1	30K	124	22
	1.5R3	39K	166	98
	1.5R4	41K	180	35
	1.5R5	44K	181	39
	1.6R1	51K	178	37
	1.6R4	51K	180	138
	1.6R5	51K	124	37

Table B.3 Projects analyzed to study LAs and developers' perception.

Project	Version	Size (LOC)	Language
ArgoUML	0.10.1 0.34	82K 195K	Java
Cocoon	2.2.0	60K	Java
Eclipse	1.0	475K	Java
Apache Maven	3.0.5	71 K	Java
Apache OpenMeetings	2.1.0	52 K	Java
GanttProject	05/2013	57 K	Java
boost	1.53.0	1.9 M	C++
BWAPI	05/2013	118 K	C++
CommitMonitor	1.8.7.831	148 K	C++
OpenCV	05/2013	544 K	Java, C++

ANNEXE C

Survey on Identifier Renaming

Tables C.1, C.2, and C.3 report the questions used for the survey on identifier renaming (Chapter 9).

Table C.1 Survey questions—part I.

q1: How often do you rename? (Single choice) ◯ Never ◯ Rarely (up to once per month) ◯ Occasionally (few times per month) ◯ Frequently (few times per week) ◯ Very frequently (almost every day) ◯ Other ___
q2: Justification/comment on the frequency of identifier renaming: (Free-form)
q3: When do you rename? (Single choice) ◯ When changing the functionality ◯ When adding new functionality ◯ When understanding code ◯ When fixing a bug ◯ When performing refactoring ◯ Apart from other development activities ◯ Other ___
q4: Justification/comment on when do you rename: (Free-form)
q5: Do you think that identifier renaming has a cost? (Single choice) ◯ No (identifier renaming is straightforward) ◯ In most cases no ◯ Sometimes no, sometimes yes ◯ In most cases yes ◯ Yes (identifier renaming requires time and effort) ◯ Other ___
q6: Justification/comment on cost of identifier renaming: (Free-form)
q7: From your experience, can you remember a case where you decided not to perform a renaming? If yes, why? (Multiple choice) ☐ High effort required ☐ High impact on the system ☐ Too risky (could introduce bugs) ☐ Potential impact on other systems using this system (e.g., as a library) ☐ Other ___
q8: Please describe the experience (if any): (Free-form)

Table C.2 Survey questions—part II.

q9: Do you use automatic tool support for identifier renaming? (Single choice) ◯ Yes, I use a tool integrated in my IDE ◯ No, I manually search and replace occurences of the old name ◯ Other ___
Would you rename an entity if: (Single choice, 5-point Likert scale: Definitely/Probably No, Undecided, Definitely/Probably Yes) **q10**: the name and functionality are not consistent? **q11**: the name does not follow the language naming conventions? **q12**: the name does not follow the team naming conventions? **q13**: the name contains an abbreviation/acronym? **q14**: the name contains a spelling error? **q15**: the name contains misleading/hard to understand words? **q16**: the name contains a negation (e.g., `notOpen`)?
q17: Other situations in which you would (or not) rename an entity: (Free-form)
Suppose you would like to rename an entity. Would the following factors impact your decision whether you perform the identifier renaming: (Single choice, 5-point Likert scale: Definitely/Probably No, Undecided, Definitely/Probably Yes) **q18**: you are not the owner of the code? **q19**: the entity being renamed is used in many places in the code? **q20**: the entity being renamed is used in other projects? **q21**: you are close to a release deadline? **q22**: insufficient (or lack of) domain knowledge?
q23: Other factors which you would consider (or not) when you want to rename an entity: (Free-form)
q24: Any additional comment you would like to share: (Free-form)

Table C.3 Survey questions—part III.

q25: Do you think that recommending identifier names/renamings (and therefore suggesting a better name from the beginning or at the time of renaming) is useful? (Single choice, 4-point Likert scale: Definitely/Probably No, Definitely/Probably Yes)
q26: Justification/comment on the on recommending identifier naming/renaming: (Free-form)
With respect to the kind of entity being renamed, is it useful to recommend identifier names/renamings (and therefore suggesting a better name from the beginning or at the time of renaming) when they are performed for (Single choice, 5-point Likert scale: Definitely/Probably No, Undecided, Definitely/Probably Yes) **q27**: Class/Interface **q28**: Attribute **q29**: Constructor **q30**: Getter/Setter **q31**: Other methods (excl. getters/setters and constructors) **q32**: Parameter **q33**: Local variable
With respect to renamings where the meaning is preserved, is it useful to recommend the following types of identifier names/renamings (and therefore suggesting a better name from the beginning or at the time of renaming): (Single choice, 5-point Likert scale: Definitely/Probably No, Undecided, Definitely/Probably Yes) **q35**: Regarding synonyms **q36**: Regarding typos **q37**: Regarding the expansion of a word **q38**: Regarding the abbreviation of a word
With respect to renamings NOT preserving the meaning, is it useful to recommend (and therefore suggesting a better name from the beginning or at the time of renaming) the following types of identifier names/renamings: (Single choice, 5-point Likert scale: Definitely/Probably No, Undecided, Definitely/Probably Yes) **q39**: regarding words with opposite meaning? **q40**: regarding words with unrelated meaning? **q41**: regarding more specific name? **q42**: regarding more general name? **q43**: regarding adding a meaning? **q44**: regarding removing a meaning?
q45: Any additional comment you would like to share: (Free-form)