Impact of Linguistic Anti-patterns on Program Comprehension

Farideh Sanei

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

August 2024

© Farideh Sanei, 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Farideh Sanei**

Entitled: **Impact of Linguistic Anti-patterns on Program Comprehension**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Diego Elias Costa

______________________________ Examiner
Dr. Emad Shihab

______________________________ Supervisor
Dr. Yann-Gaël Guéhéneuc

Approved by ______________________________
Dr. Joey Paquet, Chair
Department of Computer Science and Software Engineering

____________ 2024

____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Impact of Linguistic Anti-patterns on Program Comprehension

Farideh Sanei

Understanding a software system is a major activity during the maintenance and evolution of the system. Researchers studied how developers understand source code and which factors reduce program comprehension. They reported that shorter identifier names have a negative impact on program comprehension. The use of short identifiers is a symptom of a broader naming problem. Consequently, researchers introduced linguistic anti-patterns (LAs) to describe bad practices in the naming, documentation, and implementation of code entities, which can have a negative impact on program comprehension.

In this study, we make the hypothesis that LAs have a small impact on program comprehension because they can be easily recognised by developers. To test our hypothesis, we conducted an empirical study to investigate whether the occurrences of LAs do indeed affect code comprehension by asking participants to perform comprehension tasks on code containing or not occurrences of LAs, with and without prior knowledge of LAs. We also examined whether participants could recognise LAs by identifying them in various code snippets and assessed the ease of LA recognition with and without prior knowledge. We also evaluate the impact of knowledge of LA on program comprehension and LA recognition. We measured participants' performance (1) in comprehension tasks using the correctness of their answers, time spent, and perceived effort and (2) in recognition tasks also using correctness. Our findings indicate that, while LAs do impact program comprehension, this impact is small or negligible because LAs are recognised by developers. They also show that learning about LAs can mitigate their negative impact by improving their recognition rates.

We thus conclude with the possible negative result that linguistic antipatterns may not actually function as antipatterns, given their minimal impact on program comprehension.

Acknowledgments

I would like to express my gratitude to those who have supported me during the completion of this thesis. Their guidance and encouragement have been essential throughout this process.

I extend my sincere thanks to my supervisor, Dr. Yann-Gaël Guéhéneuc, for his invaluable guidance and support throughout my thesis. His expertise has been instrumental in shaping my research journey.

I also want to express my deepest gratitude to my husband, Farhad, for his unwavering encouragement, understanding, and support during the ups and downs of this time. His love has been my source of strength.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Overview 1

1.2 Research Methodology 5

1.3 Research Contributions 7

1.4 Thesis Outline 8

2 Related Work 9

2.1 Definition of LAs 9

2.2 Detection of LAs 10

2.3 Impact of LAs 15

2.4 Conclusion 16

3 Study Background 17

3.1 Subject Systems 17

3.2 Studied LAs 18

3.3 Detection 24

4 Experiment Design 26

4.1 Experimental Process 26

4.2 Questionnaires 27
4.3 Questions 28
4.4 Variables 29
4.5 Participants 31
4.6 Collected Datasets 32
4.7 Hypotheses 33

5 Results **35**
5.1 First Set of Research Questions 35
5.1.1 RQ1: How do LAs affect developers' program comprehension? 36
5.1.2 RQ2: How does knowledge of LAs affect developers' program comprehension? 39
5.2 Second Set of Research Questions 42
5.2.1 RQ3: What is the recognition rate of LAs by unknowledgeable participants? 42
5.2.2 RQ4: What is the recognition rate of LAs by knowledgeable participants? . 43
5.3 Conclusion for First and Second Sets of Research Questions 44

6 Discussion **46**
6.1 Linguistic Anti-patterns 46
6.2 Extended Analysis: Recognition Rates of Additional LAs (D1, F1, F2) 48
6.3 Mitigating Factors 50
6.3.1 Age, Gender and English proficiency level 52
6.3.2 Programming knowledge 52
6.3.3 Degree 52
6.3.4 Work experience 53
6.4 Threats to Validity 53

7 Conclusion and Future Work **58**

Appendix A BeforeLecture-GroupA **60**

Bibliography **69**

List of Figures

Figure 3.1 Detected LAs in the subject systems 19

Figure 4.1 Experimental Process 26

Figure 5.1 Number of Correct Responses to the quiz questions 39

Figure 5.2 Evaluation of the knowledge of LAs 40

Figure 6.1 LA Recognition Rates (Before and After Lecture) 51

List of Tables

Table 2.1 List of Linguistic anti-patterns Arnaoudova et al. (2016). 11
Table 2.2 Detected occurrences of LAs studied in Arnaoudova et al. (2016) and developers' perceptions 12
Table 3.1 Studied Systems and Versions 18
Table 4.1 Experiment Datasets ("Correct" indicates the code snippets do not contain any LA.) 33
Table 4.2 Hypotheses 34
Table 5.1 Hypotheses Results for H1 36
Table 5.2 Hypotheses Results for H2 36
Table 5.3 Hypotheses Results for H3 36
Table 5.4 Hypotheses Results for H4 37
Table 5.5 Hypotheses Results for H5 37
Table 5.6 Hypotheses Results for H6 38
Table 5.7 Hypotheses Results for H7 38
Table 5.8 Hypotheses Results for H8 41
Table 5.9 Resluts of First and Second Sets of RQs 45
Table 6.1 The Correctness Percentage for Different LAs (BeforeLecture and AfterLecture) 50

Chapter 1

Introduction

In this chapter, we introduce the main concepts, objectives, and contributions of this work.

1.1 Overview

Software anti-patterns, in contrast to design patterns (Gamma et al., 1995; Arnaoudova et al., 2013) are "poor" practices in the design, documentation, or implementation of software artifacts. They are usually introduced by developers who are not familiar with the software system at hand and–or do not have enough knowledge and experience in solving some particular problems.(Brown et al., 1998; Khomh et al., 2012).

Brown, in his book, noted that design anti-patterns can also result from the misapplication of "good" patterns in the wrong context. He introduced 40 anti-patterns that describe the most common pitfalls in the software industry. He explained that anti-patterns serve as a strategy to map a common problem with a straightforward but poor solution. He presented each anti-pattern following a template that shows the kinds of problems it deals with, indicating the symptoms of the problem and explaining the typical reasons behind it (Brown et al., 1998).

Design anti-patterns, as poor solutions to recurring design problems while designing or implementing a software system, negatively impact a system by making it more change-prone and–or fault-prone (Khomh et al., 2012). In the study by (Abbes et al., 2011), several examples of antipatterns are highlighted:

- An example of an antipattern is the Blob, also referred to as God Class. The Blob is notable for its large and complex structure, serving as a hub for a segment of a system's behavior while relying solely on other classes for data storage, typically referred to as data classes. Key features of the Blob class include its large size, low cohesion, and its reliance on data classes primarily for field storage and accessors.

- Another example of an antipattern is the Spaghetti Code, reflective of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and structure may suggest procedural programming. They may not fully leverage object-oriented mechanisms like polymorphism and inheritance, potentially hindering their effective use.

Previous work reported that design anti-patterns have a negative impact on program comprehension (Abbes et al., 2011). Hofmeister et al. (2017) also showed that shorter identifier names impact program comprehension negatively. Therefore, understanding anti-patterns, as emphasized by (Bogner et al., 2021), can help reduce the technical debt accumulated in the projects.

Linguistic anti-patterns (LAs) (Arnaoudova et al., 2016) are anti-patterns that pertain to inconsistencies in the naming, documentation, and implementation of source-code entities. These antipatterns affect the complexity of a program making it difficult to understand and maintain.

Indeed, researchers found that the identifier lexicon has a direct impact on software understandability and reusability and, thus, on the quality of the final software product (Palma et al., 2015). In this regard, researchers empirically explored the impact of identifiers and comments on source code comprehension. Their investigations reveal that programs with comments and those containing full-word identifiers are more understandable (Arnaoudova et al., 2016).

Brooks, in his theory on understanding programs, considers identifiers and comments as internal indicators for the meaning of a program. He describes program comprehension as a top-down hypothesis-driven process, starting with a broad and general hypothesis based on the programmer's knowledge of the domain. This hypothesis gets refined to more narrow hypotheses using information from the program lexicon, with developers checking details like comments against the code.

In this theory, Brooks warns that comments and code might conflict, and when that happens,

the decision on what to trust (comments or code) depends on the overall support of the hypothesis being tested rather than the type of the indicator itself. This means different developers might trust different indicators when facing contradictions among code and comments, leading to different interpretations of a program (Arnaoudova et al., 2016; Brooks, 1983). Therefore the existence of inconsistencies has the potential to misguide developers, causing them to form wrong assumptions about the behavior of the code and spend more time and effort to comprehend the source code(Arnaoudova et al., 2016).

Arnaoudova et al. (2013) presented a catalog of LAs (in six categories named simply A, B, C, D, E, and F) which we summarise in the following with short definitions and examples:

- **A. Do more than they say**, applicable to methods, contains four LAs that do more than what their signature and–or documentation indicates. For example, "Set method returns" identifies setter methods that return a value, which deviates from the conventional practice of setters not returning anything.

- **B. Say more than they do**, applicable to methods, includes five LAs that do less than what their signature and–or documentation indicates. For example, "Get method does not return" highlights getter methods that, contrary to their name, do not return anything, creating a discrepancy between the method name and its actual behavior without proper documentation.

- **C. Do the opposite of what they say**, applicable to methods, includes two LAs that describe method implementations contradicting their signatures and–or documentations. For example, "Method name and return type are opposite" identifies methods having a name inconsistent with their return type (e.g., a method named `Disable` but having `ControlEnableState` as return type (Arnaoudova et al., 2013)).

- **D. Contain more than they say**, applicable to attributes whose declaration implies that they contain more than what their signature and documentation indicate. For example, "Name suggests Boolean but the type does not" in which the attribute's name suggests less (showing a single instance) than what their type implies (storing a collection of objects).

- **E. Say more than they contain**, applicable to attributes whose declaration implies that they

contain less than what their signature and documentation indicate. For example, "Says many but contains one" implies situations where the attribute name is plural while contradicting its single type, such as an attribute named statistics with a Boolean type.

- **F. Contain the opposite**, applicable to attributes, describes situations where an attribute declaration or definition is the opposite of what their signature and–or documentation indicates. For example, "Attribute name and type are opposite" describes attributes in which names contradict their types (e.g., having antonyms `Start` as part of the attribute name and `End` forming part of its type).

Fakhoury et al. (2018a) studied the effect of LAs and readability on developers' cognitive load using a minimally-invasive functional brain-imaging technique (fNIRS) and an eye-tracker. They reported that LAs have a negative impact on developers' cognitive load. Arnaoudova et al. (2016) examined the developers' perception of the quality of code snippets containing LAs and reported that developers consider LAs to be poor practices that should be refactored.

Researchers have applied AI for linguistic anti-pattern (LA) detection in recent years. In their work, Borovits et al. Borovits et al. (2020) tackle the challenge of linguistic anti-patterns in Infrastructure as Code (IaC), addressing issues that impede code readability and maintainability. They propose an automated approach named DeepIaC, utilizing word embeddings and deep learning to identify LAs in IaC scripts, with experiments demonstrating an accuracy range of 0.785 to 0.915. Subsequently, Borovits et al. Borovits et al. (2022) introduce FindICI, a novel method employing machine learning, word embedding, and classification algorithms to detect linguistic inconsistencies in IaC scripts. The study evaluates FindICI's effectiveness by focusing on discrepancies between the logic and brief textual descriptions of IaC code units. This examination leads to the successful identification of linguistic inconsistencies, achieved through the utilization of classical and deep learning algorithms as well as various word embedding methods.

Previous work introduced LAs and studied their impact on program comprehension. However, this previous work had limitations (e.g., the number of participants in their experiments) and could not show a strong negative impact of LAs on program comprehension, e.g., (Arnaoudova et al., 2016; Guerrouj et al., 2015).

Thesis Statement

In this thesis, we propose and evaluate the impact of the presence of linguistic antipatterns (LAs) and the knowledge of LA on program comprehension, we also evaluate the relation between the impact of LA and recognition of LAs, furthermore, we assess the impact of knowledge of LA on LA recognition rate.

1.2 Research Methodology

Based on our previous work and experience in researching and teaching patterns, including LAs, we hypothesise that H0: LAs have a small impact on program comprehension because they are easily recognised as such by developers, who are thus not negatively impacted by their occurrences in their code. To test our hypothesis, we performed experiments divided into **two sets**. Our experiments involve a total of 7 LAs and 229 participants. They distinguish between participants without knowledge of LAs (in an experiment called BeforeLecture) and those with knowledge of LAs (in an experiment called AfterLecture).

In the first set about comprehension, we ask two research questions about the impact of occurrences of four LAs and the knowledge of LAs on program comprehension, before and after lecturing participants about these LAs. The LAs considered include A2 ("Is" returns more than a Boolean) and E1 (Says many but contains one), before giving a lecture on LAs to the participants, and A3 ("Set" returns a value) and B4 (Not answered question), after the participants received a lecture on LAs. In the second set about recognition, we asked two research questions about the recognition of these LAs by participants, again before and after lecturing participants about these LAs.

Thus, in this study, we ask four research questions divided into two sets:

- **First Set of Research Questions: Impact of LAs and LA Knowledge on Program Comprehension with and without LA Knowledge.**
  - **RQ1: How do LAs affect developers' program comprehension?** To answer this research question, we ask the participants to perform two comprehension tasks, using different code snippets that may or may not contain LAs. We examine whether the

occurrence of LAs in the code affects the time spent by the participants performing the tasks, their effort, and the correctness of their answers. We show that the presence of LA has small or negligible impacts on participants' program comprehension in terms of correctness and time.

    - **RQ2: How does knowledge of LAs affect developers' program comprehension?** We investigate if knowing LAs helps participants understand code snippets when performing comprehension tasks. We evaluate the participants' knowledge of LAs through a questionnaire. Then, we investigate potential correlations between their level of knowledge of LAs and their program comprehension, measured in terms of correctness, time, and effort. We show that training LA improves participants' program comprehension in terms of correctness in the presence of LA.

- **Second Set of Research Questions: Recognition rate of different LAs with and without LA Knowledge.**

    - **RQ3: What is the recognition rate of LAs by unknowledgeable participants?** For each type of LA under study, we define one detection task and study how different types of LAs affect the participants' recognition when the participants do not have knowledge of LAs (*BeforeLecture*).

    - **RQ4: What is the recognition rate of LAs by knowledgeable participants?** We repeat the same experiment as in **RQ3** but after giving a lecture about LAs to the participants (*AfterLecture*). Thus, we investigate whether knowing LAs changes the recognition rate of LAs compared to the results of **RQ3**. We show an improvement in LA recognition rates after learning about them.

The answers to these research questions show that A2 has a statistically significant impact with a small effect size because it is harder to recognise, with a 35.2% rate of correctly recognised occurrences before being taught about LAs. E1 also has a statistically significant impact but with a negligible effect size because it is easier to recognise, with a 63.63% recognition rate. After training, all the recognition rates are very high, and neither A3 (with a recognition rate of 94.8%) nor B4 (with

a recognition rate of 97.59%) have a statistically significant impact on program comprehension.

The results for the first set also show that teaching about LAs reduces the negative impact of the LAs on the participants' program comprehension in terms of correctness with statistical significance from small/negligible to non-significant. After training participants about LAs, participants have, on average, more correct answers to the questions involving LAs (97%) compared to when they did not know LAs (48%). The results for the second set also show an improvement in the recognition rate after learning about LAs. After training participants about LAs, participants could recognise them with more than 90% recognition rate.

Thus, we provide evidence supporting our hypothesis: LAs do have an impact on program comprehension but this impact is mitigated (small or negligible) when the LAs are recognised by developers. We also conclude that learning about LAs can reduce their negative impact by improving their recognition rate.

In our study, we also controlled for other factors, such as participants' experience. Finally, we studied the recognition of three other LAs, D1, F1, and F2, to understand whether the recognition rates found for the other four LAs were representative of other LAs. We observed that these three LAs have low recognition rates before training and we conclude that future work should study their impact on program comprehension.

1.3 Research Contributions

With the proposed research method, we make the following contributions:

- **Impact of LAs on program comprehension**: By analyzing data from comprehension questions, we report that the presence of LAs in a code snippet has a statistically significant negative impact on correctness and time although with small or negligible effect sizes.

- **Impact of LAs knowledge on program comprehension**: By analyzing data from comprehension tasks comparing the participants' performance before and after learning about LAs, we show that acquiring knowledge of LAs significantly improves correctness, from LA presence having small or negligible impact to not having an impact on program comprehension.

This difference and improvement between the performance of participants with and without LA knowledge is also statistically significant with a medium to large effect size.

- **Recognition rate of different LAs**: By analyzing data of detection tasks, we show that the impact of LAs on program comprehension is proportional to their ease of recognition. The easier the LAs are to recognize, the less impact they have on program comprehension.

Our contribution is under review for publication as follows:

Farideh Sanei, Zeinab (Azadeh) Kermansaravi, Yann-Gaël Guéhéneuc, Fabio Petrillo; A Large-scale Empirical Study of the Impact of Linguistic Anti-patterns on Program Comprehension; Submitted to Information and Software Technology, 2024.

1.4 Thesis Outline

This thesis is comprised of seven chapters. The rest of the study is organized as follows. Chapter 2 discusses the related literature and provides background information about LAs. Chapter 3 describes the used systems, studied LAs, and our detection approach. Chapter 4 describes the design of our experiments. Chapter 5 presents our results. Chapter 6 discusses these results and threats to the validity of our study. Finally, Chapter 7 concludes the study and outlines some avenues for future work.

Chapter 2

Related Work

Multiple studies on source-code identifiers (Caprile and Tonella, 2000; Merlo et al., 2003; Caprile and Tonella, 1999; Anquetil and Lethbridge, 1998) highlighted the necessity of choosing appropriate and meaningful identifiers during the implementation of a software system. In a related context, recent research conducted a systematic literature review on code comment quality assessment, identifying four primary quality attributes that most researchers prioritize, with a significant emphasis on ensuring consistency between comments and code (Rani et al., 2023). This chapter provides an overview of previous research concerning the *definition of LAs*, their *detection*, and their *effect* on maintenance and evolution.

2.1 Definition of LAs

Abebe et al. in (Abebe et al., 2009) proposed the first definition of lexicon bad smells. These lexicon bad smells are characterized by the use of abbreviations, contractions, and–or strange grammatical structures. They report that such lexicon bad smells have a negative impact on concept location activities. Following this pioneering work, (Arnaoudova et al., 2013) defined a new category of smells, linguistic anti-patterns (LAs), related to poor practice in naming, documentation, and implementation of an entity. They focused on a higher level of detail related to inconsistencies between method names, parameters, return types, and comments, and also between attribute names, types, and comments.

Arnaoudova et al. in (Arnaoudova et al., 2013) organized LAs in two categories based on affected artifacts, i.e., methods and attributes. Table 2.1 summarises these LAs providing their names, categories, and short descriptions.

Arnaoudova et al. in (Arnaoudova et al., 2016) examined LAs from seven open-source systems written in C++ and Java to understand how they are perceived by developers. They collected the opinions of both internal developers (who wrote the code) and external developers who had no prior knowledge of the studied systems. Table 2.2 summarises their findings. Results indicate that the majority of developers view LAs as poor practices and awareness of LAs prompts code changes in %10 of cases.

This previous work provides definitions for the linguistic antipatterns, which are poor naming practices. We build on these definitions to study tools that could detect these antipatterns.

2.2 Detection of LAs

Researchers in (Tan et al., 2007, 2011, 2012) introduced three techniques to find inconsistencies between code and comments called @*iComment*, @*aComment*, and @*tComment*. iComment combines Natural Language Processing, Machine Learning, Statistics, and Program Analysis, to automatically extract implicit program rules from comments and detect inconsistencies, revealing bugs or bad comments.

De Lucia et al.in (De Lucia et al., 2011) proposed a tool called COCONUT to detect inconsistencies between the lexicons of the high-level artifacts describing a software system. The tool compares the textual similarity of two different artifacts to find inconsistencies.

Researchers in (Abebe and Tonella, 2013) proposed an automatic approach to help developers choose proper identifiers based on the concepts used in a system. The suggested identifiers are ranked based on contextual relevance, aiding developers in writing concise and consistent identifiers. The study, conducted through simulations, demonstrates the effectiveness of the approach in providing completion suggestions that align with developers' choices, particularly with methods like term prefixes and neighboring concepts. Results suggest the applicability and potential usefulness of the proposed identifier suggestion approach in improving code quality and understanding.

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

Table 2.1: List of Linguistic anti-patterns Arnaoudova et al. (2016).

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

Table 2.2: Detected occurrences of LAs studied in Arnaoudova et al. (2016) and developers' perceptions

In (Arnaoudova et al., 2016), authors introduced Linguistic Anti-pattern Detector (LAPD) as an extension of CheckStyle, a tool to find inconsistencies among identifiers, source code, and comments. This tool is available online[1]. They discussed general naming and commenting issues and provided a catalog covering all possible LAs in methods and attributes (code and comments).

Researchers in (Fakhoury et al., 2018b) investigate the impact of poor source code lexicon and readability on developers' cognitive load, employing functional Near Infrared Spectroscopy (fNIRS) and eye tracking during software comprehension tasks. Emphasizing the critical role of lexicon quality in program understanding, the study finds a significant increase in cognitive load when linguistic antipatterns are present in the source code. The research contributes empirical evidence to the link between lexicon quality and developers' ability to comprehend software, shedding light on the importance of improving source code readability for efficient software maintenance.

Borovits et al. in (Borovits et al., 2020) performed the LAs detection in infrastructure as code (IaC) scripts used to provision and manage computing environments. They detected inconsistencies between the logic/body of IaC code units and their names using an approach that employs word embeddings and deep learning techniques. They showed that the proposed approach obtains an accuracy between 0.785 and 0.915 in detecting inconsistencies.

In (Borovits et al., 2022), the authors introduced FindICI, a novel approach that employs machine learning, word embedding, and classification algorithms to identify linguistic inconsistencies in scripts for Infrastructure-as-Code (IaC). By focusing on discrepancies between the logic and short text names of IaC code units, the study evaluates FindICI's effectiveness with experiments on Ansible tasks from open-source repositories. Results showed that both classical and deep learning algorithms, alongside various word embedding methods, successfully identify linguistic inconsistencies in IaC scripts.

Al Madi in (Al Madi, 2022) addresses the significance of identifier naming in program comprehension and the potential hindrance caused by similar identifier names. The author introduces an open-source tool, Namesake, designed for Python programs, which detects confusing naming combinations based on orthography, phonology, and semantics. By assessing similarity and flagging problematic identifier combinations, Namesake provides programmers with a valuable resource to

[1] http://www.veneraarnaoudova.com/linguistic-anti-pattern-detector-lapd/

improve naming quality, offering integration into DevOps pipelines for automated checking and appraisal, especially when combined with existing coding style checkers.

Palma et al. (2015) introduced DOLAR (Detection Of Linguistic Antipatterns in REST), an approach employing both syntactic and semantic analyses to detect linguistic (anti)patterns in RESTful APIs. They applied DOLAR to 15 widely-used RESTful APIs to identify ten linguistic (anti)patterns. The validation of DOLAR involved analyzing APIs, including those of Facebook, Twitter, and YouTube. The results showed high accuracy in detecting linguistic antipatterns (LAs) and indicated the presence of LAs in most of the analyzed RESTful APIs.

Blasi and Gorla (2018) introduced RepliComment, a tool for Java code that automates the detection and classification of source comment clones. It detects copy-and-paste errors in comments, highlights clones indicative of poorly written comments, and points developers to which comments should be fixed. They discovered over 11K comment clones in 10 Java projects, with 1,300 potentially critical cases. Manual inspection of 412 RepliComment-identified issues showed a 79% precision in critical clone detection. Further, examining 200 clones filtered as legitimate revealed no false negatives.

Palma et al. (2022) investigated the linguistic quality of REST APIs for IoT applications. Proposing the SARAv2 (Semantic Analysis of REST APIs version two) approach, they conducted comprehensive syntactic and semantic analyses of 19 REST APIs for IoT applications. They focused on the detection of linguistic patterns and antipatterns, leading to the development of the REST-Ling tool, a Web application that automates the identification process. Empirical validation using 1,102 URIs showcased the tool's accuracy which is more than 80%. The average detection time for linguistic antipatterns is short, i.e., 8.396 seconds. Also, the results revealed that linguistic antipatterns are common, and the tool effectively identifies them.

This previous work showed that it is possible to detect occurrences of LAs in software systems. We use the LAPD rules to build our detection tool.

2.3 Impact of LAs

Authors of (Abebe et al., 2011) explored the impact of lexicon bad smells, such as extreme contractions and inconsistent term use, on software comprehension tasks, specifically focusing on Information Retrieval (IR)-based concept location. Through a case study on two open-source software systems, the authors find that lexicon bad smells significantly affect concept location when using IR-based techniques. The study suggests that refactoring these lexicon bad smells can have a positive impact on the task, particularly in systems with relatively low-quality lexicons. The authors plan to further investigate the impact of individual lexicon bad smells on concept location and extend their approach to other comprehension tasks, conducting empirical studies with developers to gauge their perception of improved code lexicon.

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

Guerrouj et al. in (Guerrouj et al., 2015) performed an empirical study of the impact of anti-patterns and LAs on change and fault-proneness in object-oriented systems. Results reveal that, in some cases, classes with both design and lexical smells are more fault-prone. Additionally, classes with design smells tend to be more change- and fault-prone than those with lexical smells only, suggesting potential areas for targeted refactoring efforts. The study recommends focusing on components with design smells, with future work aiming to understand the interaction between design and lexical smells through a user study involving professional developers.

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

Aghajani et al. (2018) studied the impact of LAs on developers of client projects using APIs. In the context of modern software development relying on third-party APIs, poorly designed APIs with LAs can lead to a steeper learning curve, misunderstandings, and bug-prone code. The study, conducted on 1.6k releases of popular Maven libraries and 14k open-source Java projects, along with 4.4k related questions on Stack Overflow, investigated whether developers using APIs affected by LAs are more likely to introduce bugs and ask more questions on Stack Overflow compared to those using non-affected APIs. While statistical analysis suggests some impact, qualitative analysis prompts further investigation to better understand the observed phenomenon and isolate the effects of LAs on code-related activities through controlled experiments.

This previous work showed that LAs influence software development practices and outcomes. We build on these studies to investigate the practical impact of LAs on the developers' program comprehension.

2.4 Conclusion

Previous work introduced and defined LAs as poor practices in naming, documentation, and code and showed that they can affect program comprehension, which is a crucial aspect of software quality but they did not study the effect size of the possible impact of LAs. If participants could recognise LAs, they would not be negatively impacted by LAs occurrences in the code and in case of any impact, it is probably small or negligible.

Chapter 3

Study Background

We now describe the used systems, studied LAs, and the detection approach in this thesis.

3.1 Subject Systems

Using convenience sampling (Shull et al., 2007), we chose ten systems written in Java from different domains, and different sizes: Apache Ant, Apache Hadoop, Apache commons-lang, ArgoUML (two versions), Cocoon, JFreeChart, JHotDraw, Hibernate, Rhino, and Xerces. Table3.1 describes the studied systems.

These systems are open source allowing for the replication of our study. We also chose these systems because previous studies analyzed some of them (ArgoUML, Cocoon, JFreeChart, Xereces) to study the impact of design anti-patterns on program comprehension (Abbes et al., 2011) and that of linguistic anti-patterns on change- and fault-proneness (Guerrouj et al., 2015). The last reason for selecting these systems is that they contain different types of LAs and could be easier/more complex to understand. For ArgoUML, we considered two versions, because in ArgoUML0.14 there was a LA of type "F1", but developers solved it during the software evolution. Other types of LAs were also removed during the evolution of ArgoUML0.14. Hence, we also study the last version of ArgoUML (ArgoUML0.34).

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

Table 3.1: Studied Systems and Versions

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

3.2 Studied LAs

(Arnaoudova et al., 2016) studied LAs in seven open-source systems written in C++ and Java to understand how they are perceived by developers. They collected the opinions of both internal developers (who wrote the code) and external developers (who had no prior knowledge of the systems). They showed that developers attributed LAs to maintenance activities or inattention to naming conventions/comments. About %56 of LAs were deemed removable, and internal developers had already addressed %10. The study identified the LAs that were perceived as poor by both external and internal developers, encouraging the use of recommender tools like the developed Checkstyle extension. Building on these findings, we selected the following LAs for our study:

- A2: “Is returns more than a Boolean”
- A3: “Set method returns”
- B4: “Not answered question”
- F1: “Attributes name and type are opposite”
- F2: “Attributes signature and comments are opposite”.

We chose these LAs because they were deemed the least acceptable by developers. These LAs

were considered poor practices by both external and internal developers. In addition to these five LAs, we also selected D1 and E1 which are the top two most frequent LAs in the ten subject systems:

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

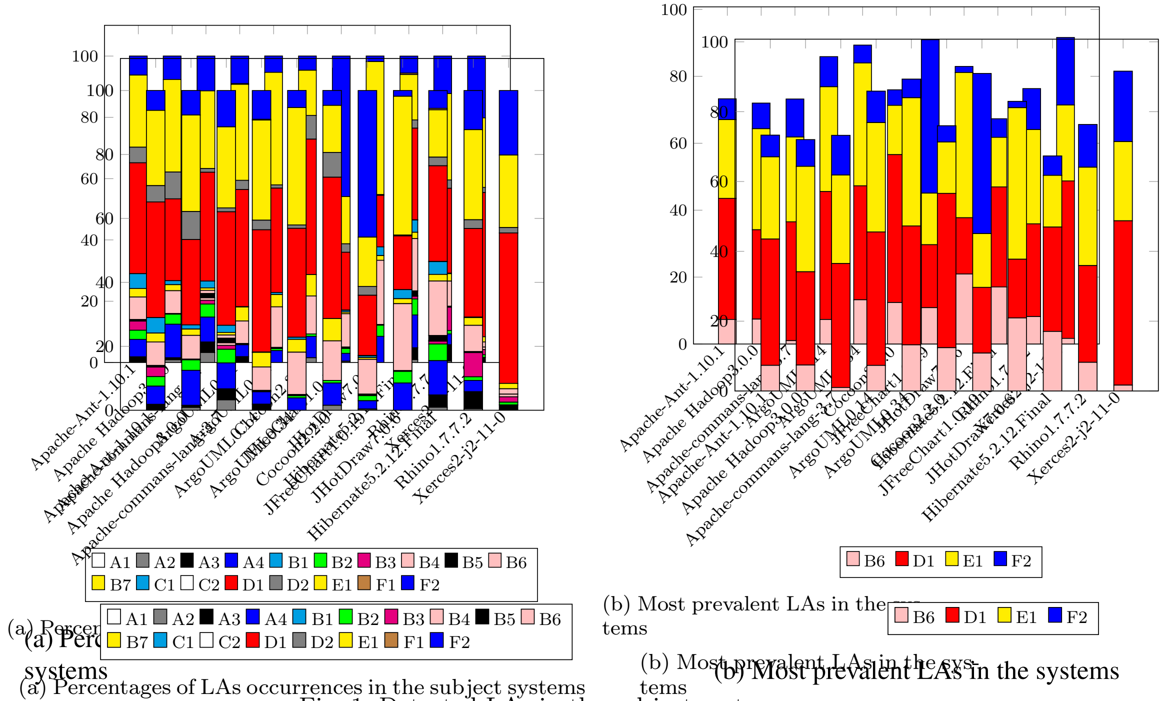


Fig. 1: Detected LAs in the subject systems

Figure 3.1: Detected LAs in the subject systems

Fig. 1: Detected LAs in the subject systems

Figure 3.1a shows the percentages of each type of LA in the subject systems. Figure 3.1b shows the most prevalent LAs in the studied systems. It shows that D1–“Says one but contains many” and E1–“Says many but contains one” are the two most frequent LAs in all the studied systems. In the following, we present some examples of the seven selected LAs to clarify their definitions. More examples of LAs are available in (Arnaoudova et al., 2013).

Listing 3.1: Example of A2–"Is returns more than a Boolean" (Cocoon2.2.0)

```
public int isValid () {
    final long currentTime = System.currentTimeMillis();
    if ( currentTime <= this . expires ) {
      // The delay has not passed yet --
      // assuming source is valid .
      return SourceValidity.VALID ;
    }
    // The delay has passed,
    // prepare for the next interval.
    this.expires = currentTime + this.delay ;
    return this.delegate.isValid ();
}
```

Listing 3.1 shows an example of "A2–"Is" returns more than a `Boolean`" LA, which belongs to the subcategory of "Methods". In this case, the method name `isValid` suggests that the method returns a `boolean` but it returns an integer, which is counterintuitive. This inconsistency arises within the class called `DelayedValidity`.

Listing 3.2: Example of A3–“Set method returns” (Apache Ant 1.10.1)

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

Listing 3.2 presents an example of LA of type “A3”, where the method name is `setProperty` but it returns `Object`. Setters return types should be `void`.

Listing 3.3: Example of B4–“Not answered question” (ArgoUML v0.34)

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

Listing3.3 presents an example of the “B4–Not answered question" LA. The method `has EditableBoundingBox` declared in class `StylePanelFig` has a name that suggests a `Boolean` value but it returns nothing (`void`).

Listing 3.4: Example of D1–"Says one but contains many" (Rhino1.7.7.2)

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

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

Listing 3.4 shows an example of LA of type "D1". The attribute `critic` has a singular name but the type is `Vector`.

Listing 3.5: Example of E1–"Says many but contains one" (Hibernate-release-5.2.12-final)

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

    public Caching(TruthValue requested){
      this.requested = requested;
    }
}
```

Listing 3.5 contains an LA of type "E1", where the name of the attribute is `cacheLazy Properties` while its type is `Boolean`. The name of the attribute suggests that it stores a collection of properties but its type is Boolean.

Listing 3.6: F1–Example of "Attributes name and type are opposite" (ArgoUML 0.14)

```
MAssociationEnd Start = null;
```

Listing 3.6 presents a LA of type "F1". The name of an attribute of class `ActionNavigability` is start, while it uses end as a part of the type of the attribute.

Listing 3.7: F2–Example of "Attribute signature and comment are opposite" (ArgoUML 0.14)

```
public class SuffixLines extends BaseParamFilterReader implements
    ChainableReader{
     // Parameter name for the prefix.
     private static final String SUFFIX_KEY = ''suffix'';

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

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

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

Listing 3.7 presents a LA of type "F2". The attribute `SUFFIX_KEY` in class `SuffixLines` is a suffix while the comment documenting it says `Parameter name for the prefix`.

3.3 Detection

LADP is a tool for identifying inconsistencies among identifiers, source code, and comments (Arnaoudova et al., 2016). It handles generic naming and comments issues in object-oriented programs, specifically in the lexicon and comments of methods and attributes. It can detect LAs in C++ and Java source code, but it is slow on large systems. We implemented an extension of PMD[1],

[1] `https://pmd.github.io/`

reusing the same algorithms as (Arnaoudova et al., 2016) to make the detection process faster. This extension is available on-line[2].

PMD is an open-source static Java code analyzer. It is a plug-in for IDEs, like Eclipse or jEdit. It is used to find bad programming practices that can reduce performance. It also detects copied–pasted pieces of code and other poor practices. It uses an Abstract Syntax Tree to identify occurrences of poor practices. We added new rules and rule sets related to each type of LA to detect linguistic anti-patterns. These rules stem from the definitions of the LAs (Arnaoudova et al., 2016).

We implemented our PMD extension to detect LAs in methods, attributes, parameters, and documentation. We used the Stanford parser (Toutanvoa and Manning, 2000) to extract the POS of words and WordNet to obtain the meaning of the words in the documentation, the methods signatures, and the relations between the words (like antonyms or synonyms).

To ensure that the occurrences of the studied LAs detected in the subject systems were true instances, we manually verified every code snippet involved in the experiments to confirm the presence (or absence) of LAs in the snippets.

[2] https://www.ptidej.net/downloads/replications/ist24b/

Chapter 4

Experiment Design

We now describe our experimental process, participants, questions, hypotheses, and independent and dependent variables.

4.1 Experimental Process

We performed two experiments to evaluate the impact of LAs on participants' program comprehension. We conducted experiments over six weeks. Figure4.1 shows our experimental process.

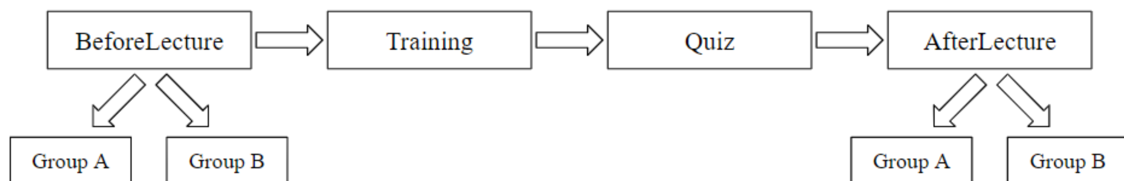


Figure 4.1: Experimental Process

The first experiment called BeforeLecture, conducted over one week, involved participants without prior knowledge of LAs to assess the impact of LAs on program comprehension. Then, we taught these participants about LAs and evaluated their knowledge of LAs through a quiz test within one week to ensure that they understood LAs well. To investigate whether knowing LAs can improve program comprehension in terms of correctness, time, and effort, two weeks later, participants had one week to perform the second experiment, called AfterLecture. These two experiments

also allowed us to evaluate the recognition rates of studied LAs among participants with and without prior knowledge of LAs and to determine whether familiarity with LAs led to improvements in their recognition rates in terms of correctness.

In BeforeLecture, we provided two similar online questionnaire forms for two groups. Group A were participants with odd student IDs while Group B were those who had even student IDs. We divide by student IDs to avoid bias (e.g., gender bias). Each group has six questions related to different types of LAs. AfterLecture has the same structure as BeforeLecture, but different code snippets, and different questions.

For each step (experiments and quiz), we asked participants to complete all the tasks within the same day of their choice, considering the fact that their time for each question is calculated using a hidden automatic timer.

4.2 Questionnaires

Each questionnaire contains four questions: Q1 and Q2 are general comprehension questions on excerpts from the studied systems; Q3 is a detection question in a Java class; Q4 includes three sub-questions Q(4.a to 4.c) to ask participants to find possible LAs in code snippets and to suggest the refactoring solutions. We provide one questionnaire in Appendix A as an example. The complete set of questionnaires for BeforeLecture, AfterLecture, and Quiz are available on-line[2].

We give Q1 to Group A with one particular type of linguistic anti-pattern while Group B is given a refactored version of the same source code (i.e., with the LA removed). The type of LA we chose for Q1 is A2 in BeforeLecture and A3 in AfterLecture. We do the contrary for Q2 in which Group B receives the LA and Group A has the refactored code. The type of LA we chose for Q2 is E1 in BeforeLecture and B4 in AfterLecture. This way, we can evaluate the impact of LAs on participants' program comprehension by comparing their comprehension when LAs are both present and absent within the same code snippets.

For the first two questions in BeforeLecture, studied LAs are A2 "Is returns more than a Boolean" and E1 "Says many but contains one" from the attribute category. In the same questions of AfterLecture, studied LAs are A3 "Set method returns" and B4 "Not answered question".

We chose them to be different from LAs in BeforeLecture to understand the impact of these two LAs on program comprehension. This choice also prevents participants from giving answers based on what they learned by experience during the first experiment, which we discuss further in Section6.4 related to the threats to validity.

For BeforeLecture, in the first question (Q1), we asked participants to find the method name that does a specific task to understand whether bad method naming can impact their understandability, and for the second question (Q2), we asked them a similar question, but now focusing on variables, to understand how bad attribute naming can affect their understandability. We asked participants in Q3, which element(s) were responsible for confusion (in case they reported to have been confused by some identifiers and–or comments). Q(4.a to 4.c) were about finding bad coding practices in some code snippets and comments. In AfterLecture, for both questions (Q1, and Q2), we asked comprehension questions on code containing “A3”, and “B4”, to understand the impact of these two LAs on code understandability. Q3 and Q(4.a to 4.c) were similar to those asked in BeforeLecture.

4.3 Questions

We used comprehension and detection questions to collect data on the participants’ performances. We selected the Java classes and code snippets from the studied systems based on the studied LAs and ensured that each Java class and code snippet contained only one type of LA. We also produced refactored versions of these code snippets (containing no LA).

Similarly to (Abbes et al., 2011), we considered questions in two of the four categories of questions regularly asked and answered by developers (Sillito et al., 2008):

- Finding a focus point in some subset of the classes and interfaces of some source code, relevant to a comprehension task;
- Focusing on a specific class that is believed to be associated with some particular task and on directly related classes;
- Understanding a number of classes and their relations in some subsets of the source code;
- Understanding the relations between different subsets of the source code.

We only chose questions in the first two categories, because the last two categories include questions that depend on different subsets of the source code, and, in our experiments LAs are located in a single subset of the code (i.e., they do not span multiple classes).

For each group of participants, we defined the following two categories of questions:

- Category 1: Finding focus points:
  - (1) Which method is used to determine **this value**?
  - (2) How does this assignment affect **this variable**?
- Category 2: Expanding focus points:
  - (1) Where is the bad coding practice (respectively LA) in **this method**?
  - (2) Where is the source of confusion in **this method**?

The text in bold is a placeholder that we replaced with appropriate behaviors, concepts, elements, methods, and types depending on the systems on which the participants were performing their tasks. For example, in the case of Apache Ant, we replace "this value" in Question 1, Category 1, by `restricted` and the question reads: "Which method is used to determine the value of the `restricted` attribute in the following link (Click on the link to show the snippet code)?".

After designing the survey questionnaires, to evaluate our questions we conducted a pilot study with five students from our research group, who are experienced developers. Using their feedback we refined the formulation of some questions before starting our study. We ensured to ask questions that were neither too simple nor too difficult while providing the data that we needed to answer our research questions. All the questions used in our experiments are available online[2]

4.4 Variables

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

We have six mitigating factors that could influence the participants' program comprehension in terms of correctness, time, and effort, as follows:

- Participant’s age
- Participant’s gender
- Participant’s degree
- Participant’s knowledge level of programming
- Participant’s working experience
- Participant’s level of English fluency

We collected these mitigating factors using a postmortem questionnaire (feedback). This feedback questionnaire was filled out by participants at the end of BeforeLecture. We evaluate the impact of these mitigating factors on program comprehension in Section6.3.

The dependent variables measure the participants’ program comprehension in terms of the correct answers, the time spent answering the questions, and the effort spent doing the task.

We measure the participants’ effort using the NASA Task Load Index (TLX) (Hart and Staveland, 1988). TLX is a multi-dimensional measure that evaluates the participants’ subjective workload based on a weighted average of their ratings on six scales: mental demands, physical demands, temporal demands, own performance, effort, and frustration. We consider three scales including “mental demands”, “effort”, and “frustration” related to one of our metrics “Effort”. NASA provides a computer program to collect weights for the six scales and ratings on these six scales. We combine weights and ratings provided by the participants into an overall weighted workload index by multiplying ratings and weights; the sum of the weighted ratings divided by fifteen (sum of the weights represents the effort)(Hart and Staveland, 1988).

An example of questions asked from participants to capture “mental demand” is: “How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? We asked the participants to provide their responses on a scale from 1 (low) to 5 (high) for each question.

We measured the time using a hidden timer in the questionnaires (BeforeLecture and AfterLecture). The timer automatically started when the participant began performing the comprehension task and stopped when the participant finished the task and submitted an answer. We received an

automatic email whenever a participant completed a task. The email contains their answers to the tasks, the time they spent on the tasks, and their reported effort (from TLX). To detect correct answers, we defined some keywords for each question and made a query based on these keywords instead of reading the whole answers.

To compute correct answers, we defined some keywords for each question and made a query based on these keywords instead of reading the whole answers. The keywords consisted of methods or attribute names that were necessary to exist in the correct answers. We initially filtered participants' answers based on the presence of these keywords. Then, we performed further evaluation on the responses passing this criterion to ensure the validity of the entire answer. We manually identified correct answers for questions not specifying method or attribute names.

4.5 Participants

A total of 229 participants were involved in our experiments. These participants were students from Polytechnique Montréal (PM) and Concordia University (CU). The participants from PM were students attending the courses LOG8430 (Software Architecture and Advanced Design) and LOG8371 (Software Quality Engineering), while those from CU were enrolled in the course SOEN6461 (Software Quality Engineering). Most of the participants reported having good English and programming skill levels. participants were mostly men, with bachelor's degrees and aged mostly between 18 to 25. A majority of them had between 1 to 3 years of experience working in industry before their studies.

The courses LOG8430 and LOG8371 are cross-listed, and attended by both engineering students and master's students. The majority of the students from PM were engineering students. The course SOEN6461 is a graduate core course, mandatory for all graduate students enrolled in the master's program in software engineering at CU. The students from PM had extensive training in software engineering, including software quality and antipatterns and code smells while the students from CU had very diverse backgrounds, most without prior knowledge of software quality. This experiment was as an assignment for these courses and participants could get extra points for their final grades.

The number of participants in the two experiments is different and that is due to the voluntary

nature of the experiment, where participants were not forced to participate in the experiments. This variation explains the different numbers of answers per LA in BeforeLecture and AfterLecture. For example, in BeforeLecture for PM, Group A which answered a question about 'F2' consisted of 77 participants, while in AfterLecture, Group A, which worked on the same LA ('F2'), included 45 participants. We chose to keep the results of all students who participated in our experiments to ensure the precision of our findings. This is rare to have such a high number of participants, as in the field of software engineering, obtaining a large number of participants is typically challenging.

4.6 Collected Datasets

Table4.1 summarises the experiment datasets. For each experiment (BeforeLecture and AfterLecture), each question (Q1 to Q4.c), and each Group (A or B), we have a dataset. We combine these datasets into different combinations to perform our statistical analyses and test our hypotheses. The datasets collected for our hypotheses are:

- DS_{1-4} consist of data from code snippets with LA and their corresponding code snippets without LA considering each question (Q1 and Q2) and each experiment separately.
- DS_{5-6} consist of the combination of withs in Q1 and Q2 and the combination of withouts in Q1 and Q2 in BeforeLecture/AfterLecture.
- DS_7 consists of the combination of all the withs and the combination of all the withouts available in both questions of both experiments.
- DS_8 consists of the combination of withs in BeforeLecture and the combination of withs in AfterLecture in both questions.
- DS_{9-11} consist of the above datasets for PM/CU/the combination of PM and CU datasets (called Combined dataset).

Table 4.1: Experiment Datasets ("Correct" indicates the code snippets do not contain any LA.)

Question	BeforeLecture		AfterLecture	
	Group A	Group B	Group A	Group B
Q1	A2 (with)	A2 (without)	A3 (with)	A3 (without)
Q2	E1 (without)	E1 (with)	B4 (without)	B4 (with)
Q3	F1	F1	D1	D1
Q4.a	F2	D1	F2	F2
Q4.b	Correct	Correct	F1	F1
Q4.c	Correct	F2	A3	A3

4.7 Hypotheses

We formulated eight hypotheses to investigate our first set of research questions, including RQ1 (H1-H7) and RQ2 (H8), as detailed in Table4.2. We defined H1 to H7 to assess the statistical significance of differences in terms of correctness, time, and effort between participants working on code snippets with LAs and participants working on code snippets without LAs. We defined these hypotheses to answer RQ1 and see if the presence of LAs has any statistically significant impact on program comprehension. We defined H8 to study whether there are any statistically significant differences between participants who were unfamiliar with LAs and those knowledgeable about LAs in terms of correctness, time, and effort. Using this hypothesis, we can answer RQ2 on whether knowing LAs can mitigate their impact on program comprehension. These hypotheses are then tested using data collected from comprehension questions.

To ensure the validity of the results of hypotheses testing, we assess the normality of our data using the Shapiro-Wilk test. Depending on its result, we then employ a non-parametric test for non-normally distributed data and a parametric test for normally distributed data. We should then carefully consider the appropriate test, taking into account whether our compared variables are of the same size or unequal sizes. After applying the test and finding statistically significant values in our hypotheses, we should find the direction of these significant values. We calculate the average of the compared variables for this purpose.

In H1 to H7, we check all reasonable comparisons of participants' answers, their time, and effort to the questions with and without LAs in each and both experiments. Considering N=1 to 7,

we generally write our hypotheses as follows:

> HN: {Before learning, After learning, Combining both experiments}, given {Q1/Q2/Q1 & Q2}, there is no difference in terms of correctness, time, and effort between participants working on code with LAs and participants working on code without LAs.

H8 is given Q1 & Q2, there is no difference in terms of correctness, time, and effort between participants having knowledge of LA and participants not having knowledge of LA.

Table 4.2: Hypotheses

Hypothesis	**RQ Number**	**Experiment**	**Experiment Question**
H1	RQ1	BeforeLecture	Q1
H2	RQ1	BeforeLecture	Q2
H3	RQ1	BeforeLecture	Q1 & Q2
H4	RQ1	BeforeLecture & AfterLecture	Q1 & Q2
H5	RQ1	AfterLecture	Q1
H6	RQ1	AfterLecture	Q2
H7	RQ1	AfterLecture	Q1 & Q2
H8	RQ2	BeforeLecture & AfterLecture	Q1 & Q2

The results support H1 to H7 if they align with our hypotheses, showing that the presence of LA does not affect program comprehension in terms of correctness, time, and effort. The results support H8 if they indicate that having knowledge of LA does not affect program comprehension in terms of correctness, time, and effort. Conversely, if the result for H1 to H7 shows that the presence of LA affects program comprehension in terms of our metrics and the result for H8 shows having knowledge of LA affects program comprehension in terms of our metrics, we consider it a rejection of our hypothesis. We also calculate the effect size of any significant value to quantify the magnitude of the difference. The choice of effect size measure also depends on the normality of our data and the size of the variables.

Chapter 5

Results

This chapter presents the results for the first set of RQs (investigating the impact of LAs and LA knowledge on program comprehension) and the second set of RQs (evaluating the recognition rates of different LAs by unknowledgeable and knowledgeable participants).

5.1 First Set of Research Questions

In the first set of research questions including RQ1 (how do LAs affect developers' program comprehension?) and RQ2 (how does knowledge of LAs affect developers' program comprehension?), we present the results of evaluating the impact of LAs and knowledge of LAs on program comprehension by analyzing the first two comprehension questions (Q1 and Q2). We applied the Shapiro-Wilk test to our dataset and discovered that none of the collected data had a normal distribution, leading us to choose among non-parametric statistical tests for our analyses. Moreover, in some hypotheses, the variables being compared had unequal sample sizes. Based on these two observations, we chose the Mann-Whitney U test because it is a non-parametric test that can handle unequal sample sizes, with a significance level set at alpha = 0.05. For the same reasons, we chose Cliff's delta, a non-parametric measure that suits our data, to evaluate the effect size. Tables from Table 5.1 to Table 5.8 summarise the p-value computed by the Mann-Whitney U test for our hypotheses on the three datasets including Concordia, Polytechnique, and the Combined (combination of Concordia and Polytechnique) datasets. The statistically significant values are shown in bold.

The table also represents the effect size for the statistically significant values.

Table 5.1: Hypotheses Results for H1

		Correctness			**Time**			**Effort**			
		p-value	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**No. Responses**
H1	Concordia	0.12	0.5	0.65	0.30	8489.1	6968.01	0.16	2.85	3.12	100
	Poly	0.37	0.27	0.34	**0.000002**	541.62	2570.8	0.527	2.66	2.5	129
	Effect Size	-	-	-	0.49	-	-	-	-	-	-
	Combined	**0.017**	0.35	0.509	**0.005**	3346.6	4489.32	0.29	2.73	2.82	229
	Effect Size	-0.15	-	-	0.21	-	-	-	-	-	-

Table 5.2: Hypotheses Results for H2

		Correctness			**Time**			**Effort**			
		p-value	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**No. Responses**
H2	Concordia	0.70	0.58	0.54	0.45	13548.9	8842.28	**0.04**	3.20	2.83	100
	Effect Size	-	-	-	-	-	-	0.22	-	-	-
	Poly	**0.01**	0.69	0.87	0.63	111570.30	1730.38	0.92	2.71	2.70	129
	Effect Size	-0.17	-	-	-	-	-	-	-	-	-
	Combined	**0.048**	0.63	0.75	0.138	12613.5	4440.4	**0.058**	2.97	2.74	229
	Effect Size	-0.11	-	-	-	-	-	0.13	-	-	-

Table 5.3: Hypotheses Results for H3

		Correctness			**Time**			**Effort**			
		p-value	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**No. Responses**
H3	Concordia	0.392	0.55	0.61	0.8431	11423.8	7755.2	0.6615	3.06	3	200
	Poly	**0.0004**	0.44	0.65	**0.000009**	4987.2	2069.1	0.73	2.68	2.62	258
	Combined	**0.01**	0.48	0.63	**0.02**	7798	4552.1	0.60	2.84	2.78	458
	Effect Size	0.48	-	-	0.16	-	-	-	-	-	-

5.1.1 RQ1: How do LAs affect developers' program comprehension?

To answer this question, we consider hypotheses H1 to H7, which explore all possible relations among code snippets with and without LAs in BeforeLecture and–or AfterLecture. In the following, we analyse the participants' answers to each question considering the correctness of their answers, their spent time, and their effort. Tables including Table 5.1 to Table 5.7 summarise the results computed by the Mann-Whitney U test for defined hypotheses.

Correctness : The test results show that H5 to H7 lack statistical significance, whereas H1 to H4 exhibit statistically significant differences in terms of correctness. The results show that participants working on a piece of code snippet without LA had higher correctness in answering the questions compared to the others, implying a better understanding of code. As mentioned in Section4, we computed the effect size for the ones that show a significant difference. The calculated effect sizes present a large effect size for H3 and small or negligible effect sizes for the remaining three hypotheses.

Table 5.4: Hypotheses Results for H4

		Correctness			Time			Effort			
		p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	No. Responses
H4	Concordia	0.63	0.77	0.78	0.92	27868.8	24750.4	0.41	3.23	3.13	418
	Poly	**0.056**	0.68	0.76	**3.72e-07**	9160.6	2553.2	0.14	2.79	2.62	394
	Effect Size	-0.082	-	-	0.28	-	-	-	-	-	-
	Combined	**0.01**	0.69	0.77	**0.02**	16413.6	13979.8	0.57	2.95	2.88	812
	Effect Size	-0.07	-	-	0.09	-	-	-	-	-	-

Table 5.5: Hypotheses Results for H5

		Correctness			Time			Effort			
		p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	No. Responses
H5	Concordia	0.827	0.959	0.95	0.73	68844.5	52666	0.29	3.48	3.25	109
	Poly	0.22	0.97	0.91	0.27	4253.3	5800	0.11	2.42	2.78	68
	Combined	0.36	0.96	0.93	0.29	37923.2	39679.2	0.29	2.97	3.12	177

We conclude that the occurrence of linguistic anti-patterns negatively impacts the comprehension of the code snippets with small or negligible effect sizes.

Time : For each research question, we observed the same trend whether we considered outliers or not, so we chose not to remove them to have more data for our analysis. We assumed that the outliers are due to participants who did not consider the time, i.e. they opened a question and took a break. Indeed, participants did not know that the time was recorded because the timer was hidden to ensure that participants could answer the questions without feeling pressured by time constraints, fostering a more accurate and natural response.

Results obtained from performing the Mann-Whitney U test on H3 and H4 show statistically significant differences with small or negligible effect size, in terms of Time. They show that participants working on a piece of code with LA spend more time answering the questions than others. Also, there is a significant value in terms of time for H1, it rejects the hypothesis with a large effect size for Polytechnique and a small effect size for the Combined dataset by showing that participants working on a piece of code with LA spend less time answering the questions than others. When we consider the results of H3 and H4, which include more participants than H1, the results are significant. Therefore, based on the results from H3 and H4, we conclude that participants spend more time comprehending the code snippet with LA. However, the effect size is not that large.

Explanations: The inconsistency in time taken in H1 could be due to the performance of the Polytechnique population in handling code containing LA type A2. It seems that participants from Polytechnique may have experienced more confusion or challenges when answering questions having LA type A2. As a consequence, this factor has influenced the Combined data, leading to a

Table 5.6: Hypotheses Results for H6

		Correctness			Time			Effort			
		p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	No. Responses
H6	Concordia	0.45	0.98	0.95	0.39	21813	25252.2	0.96	3.31	3.28	109
	Poly	0.17	0.95	1	0.68	199	2281.1	0.19	2.91	2.55	68
	Combined	0.90	0.975	0.978	0.16	15824.1	14255.4	0.13	3.2	2.93	177

Table 5.7: Hypotheses Results for H7

		Correctness			Time			Effort			
		p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	p-value	Avg. With	Avg. Without	No. Responses
H7	Concordia	0.47	0.97	0.95	0.95	49956.07	40342.4	0.48	3.39	3.26	218
	Poly	1	0.97	0.97	0.82	2882	3471.6	0.84	2.5	2.63	136
	Combined	0.55	0.97	0.88	27560.3	26177.37	0.76	3.08	3.02	354	177

contradiction with H1 and a reduction in the overall effect size to a small value due to its combination with the Concordia dataset.

Based on our observations, we conclude that the occurrence of LA can increase the time to comprehend a code snippet, with a small effect size.

Effort: The statistical test results on H2 indicate a statistically significant difference in terms of effort between participants dealing with the code snippet including LA and the ones without LA. It shows participants reading a piece of code snippet with LA type E1 make more effort in answering the question than the others. We obtained the effect size of small and negligible for Concordia and Combined datasets respectively.

Explanations: The reason could be related to the type of LA which shows that participants in Concordia had a difficult time finding correct answers in the code snippet containing the LA type E1 "Says many but contains one". Furthermore, the effort in H2, being the only statistically significant result for this metric, may have been produced by chance.

Our observations show that occurrences of linguistic anti-patterns do not significantly impact the effort to understand code, except for type E1 with, at most, a small effect size.

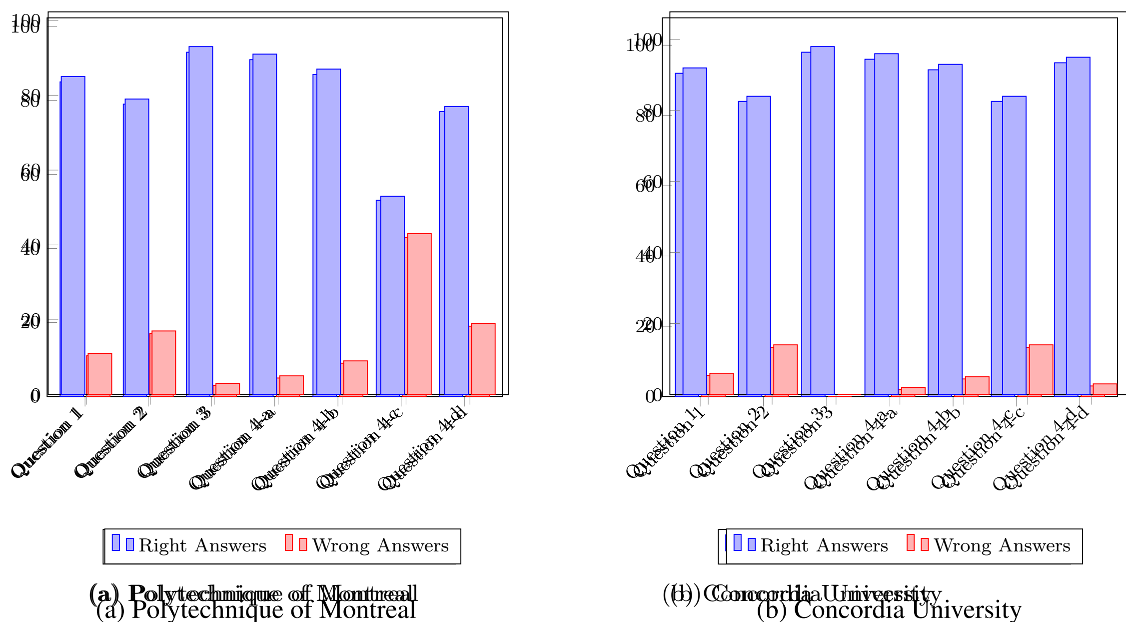
Conclusion for RQ1

Based on our observations for this research question, we conclude that LA presence increases time and decreases correctness to comprehend the code. However, the magnitude of this effect is small or negligible. This result is statistically supported for the PM dataset but not for the CU dataset.

5.1.2 RQ2: How does knowledge of LAs affect developers' program comprehension?

In this research question, we want to know if the participants' comprehension of code is proportional to the level of their knowledge (of LAs). To achieve this goal, after performing BeforeLecture, we taught participants about LAs, their consequences, and possible refactoring solutions and right after the lectures, we conducted a quiz to capture the level of knowledge of the participants. We defined our hypothesis, H8, to answer this research question.

Figure5.1b presents the number of correct answers for each question of the quiz for the participants from Concordia University. In the quiz, Questions 1, 2, and 3 are general definition questions and more than 93% of the participants answered them correctly. After these first three questions, we asked participants to detect LAs in four different code snippets containing respectively A2, A3, D1, and F1. Figure5.1b shows that more than 96% of participants could detect A2, A3, and F1 correctly. The participants were less successful in detecting D1, but still, almost 85% of participants answered the corresponding question correctly. At Polytechnique Montreal, Figure5.1a shows that more than 85% of participants could detect A2, A3, and F1 successfully. In general, all participants displayed a good knowledge of LAs after the lectures, showing that our teaching was efficient.



(a) Polytechnique of Montreal

(b) Concordia University

Figure 5.1: Number of Correct Responses to the quiz questions

Fig. 1: Right Answers (number)

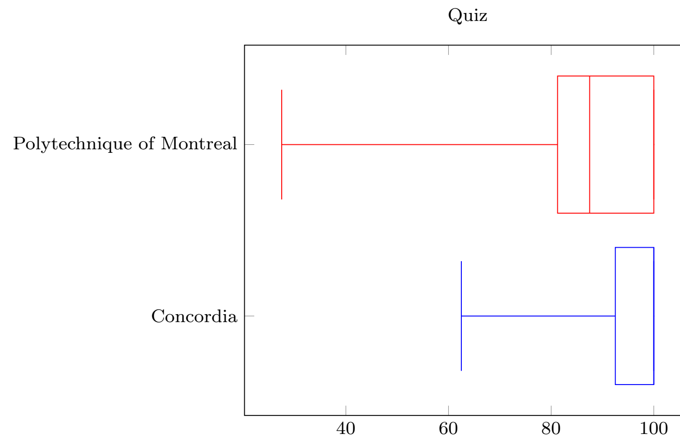


Fig. 1: Evaluation of the knowledge of LAs
Figure 5.2: Evaluation of the knowledge of LAs

Figure5.2 presents the results of the quiz for participants from Polytechnique Montreal and Concordia Universities separately. Almost all the participants could answer more than 85% of the questions correctly. Therefore, we conclude that the participants on average have a good knowledge of linguistic anti-patterns in theory and practice.

The results from RQ1 indicate an improvement in participants' performance, showing a reduction in the impact of LA presence on program comprehension. Initially, the impact was significant with small or negligible effect sizes in the BeforeLecture. However, this impact decreased to non-significant in the AfterLecture. This suggests that learning about LAs can mitigate their impact on program comprehension. We also conducted a statistical analysis to confirm the difference in participants' performance before and after acquiring knowledge about LAs.

Statistical Results: The statistical tests conducted on H8 revealed a statistically significant difference with a medium to large effect size, in terms of correctness (Table 5.8 represents the results). It shows that participants who have knowledge of LAs have more correct answers when answering the questions with LA than the participants who do not have knowledge of LAs. There is also a statistically significant difference in terms of effort between participants who have knowledge of LA and the participants who do not have knowledge of LA, although it rejects our hypothesis, it shows that when working with LAs, participants who have knowledge of LAs make more effort to answer the questions than the participants who do not have knowledge of LAs. The effect size for the CU dataset is small, the effort impact is propagated in the Combined dataset but because of its

combination with the PM dataset, its effect size is negligible in terms of effort.

The analysis of H8 shows statistically significant differences in terms of time between students at Polytechnique Montreal University, and also between students at Concordia University, with small effect sizes. Surprisingly, the results for the Concordia dataset reject the hypothesis by indicating that participants with knowledge of LA spent more time answering the questions. However, when considering the combined dataset, the effects of results for participants of CU and PM seem to neutralize each other, resulting in non-significant differences in terms of time.

Table 5.8: Hypotheses Results for H8

		Correctness			**Time**			**Effort**			
		p-value	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**p-value**	**Avg. With**	**Avg. Without**	**Responses**
H8	Concordia	**4.67e-13**	0.55	0.97	**1.23e-04**	11423.8	42956	**0.04698**	3.06	3.39	209
	Effect Size	-0.42	-	-	-0.30	-	-	-0.15	-	-	-
	Poly	**3.73e-13**	0.44	0.97	**0.003**	4987.2	2882	0.4131	2.68	2.58	197
	Effect Size	-0.52	-	-	0.25	-	-	-	-	-	-
	Combined	**< 2.2e-16**	0.48	0.97	0.17	7798.01	27560.3	**0.04**	2.84	3.08	406
	Effect Size	-0.48	-	-	-	-	-	-0.10	-	-	-

Explanations: Analyzing the performance of students at CU and PM without prior knowledge reveals interesting findings. Before learning about LA, CU students had less knowledge of patterns compared to their PM counterparts. However, after attending the lecture on LA and evaluating their answers, CU students showed remarkable progress in correctness. This progress was at the cost of more time and perceived effort, indicating their careful approach and commitment to fully understand the concepts. In contrast, PM students, who had better backgrounds because they already passed a course related to design patterns and anti-patterns (Section4.5), after the lecture showed better performance, resulting in higher speed and more accuracy in their responses.

We believe that the initial difference in familiarity with design anti-patterns and design patterns between CU and PM students influenced their performance. After learning about LA, CU students made significant improvements in correctness, although it required more time and perceived effort. Meanwhile, PM students showed a swift and more confident approach after learning.

We conclude that having knowledge of linguistic anti-patterns can mitigate the negative impact of LAs, increasing the correctness with a medium to large effect size.

Conclusion for the first set of RQs

When examining the impact of LAs, we observed that there is a statistically significant effect; however, the effect size tends to be small or negligible. This observation suggests that while LAs have a negative influence on program comprehension, it may not be substantial. However, when considering the aspect of learning about LAs, we observed a different trend. The impact on correctness is statistically significant and the effect size varies between medium and large. This finding implies that actively learning about LA has a significant positive impact on correctness, and the effect size is more substantial. Considering these findings, it would be beneficial for our community to enhance awareness and encourage the understanding of LAs among students. This can lead to students benefiting from the medium to large effect size in improved correctness and overall comprehension of code associated with learning about LAs.

5.2 Second Set of Research Questions

In this set of research questions, we investigated the recognition rate of our studied LAs by participants both before and after acquiring knowledge of LAs.

5.2.1 RQ3: What is the recognition rate of LAs by unknowledgeable participants?

In this research question, we analyze the data obtained during BeforeLecture regarding LAs type A2 and E1. In BeforeLecture, we investigated the recognition rate of these LAs by participants, in terms of correctness. We did not consider time and effort when answering this research question because the level of complexity of code snippets used in the different questions is not equivalent. Some questions include small code snippets, while others include larger code snippets. As it is expected that participants will take a longer time and make more effort inspecting a larger code snippet than a smaller code snippet, a comparison of time and effort would not have been meaningful.

A2- "Is returns more than a Boolean" Participants from Group A in BeforeLecture, answered the first question (Q1) including this type of LA. The results show that 42 participants out of 119 (35.29%), could answer the question correctly.

A2- "Is returns more than a Boolean" could be recognised with a reasonable rate.

E1- "Says many but contains one" In BeforeLecture, the second question for Group B contains E1. In total 70 out of 110 (63.63%) participants could answer the question correctly.

E1- "Says many but contains one" has been answered correctly by most of the participants indicating that E1 is easy to recognise. We investigated E1 in H2 of RQ1 and showed its negligible impact on program comprehension. It seems that the reason for its negligible impact on program comprehension is the ease of its recognition.

The results show that LA type A2 with a more significant impact (at most small) is harder to recognise and LA type E1 with negligible impact is easier to recognise.

5.2.2 RQ4: What is the recognition rate of LAs by knowledgeable participants?

To answer this research question, we do the same steps as for RQ3 but focus on the data obtained from AfterLecture. Similar to RQ3, we only consider correctness, to evaluate the participants' recognition rate. We want to know whether having knowledge of LAs can improve the LA recognition rate by participants. In the following, we discuss each type of LA in more detail, following the same format as RQ3.

A3- "Set method returns" In AfterLecture, both groups of participants had a question (Q4.c) on code containing this LA. We obtained 166 correct answers out of 177 submitted responses. Group A also had the first question (Q1) on code containing this LA. We found that 91 participants answered correctly out of 94 submitted answers. When combining the results of the two questions, we obtain that 94.83% of participants could do the task correctly.

A3- "Set method returns" has been answered correctly by more than 94% of participants showing that it is very easy to recognise.

B4- "Not answered question In AfterLecture, only participants in Group B answered one Question (Q2), which involved code containing B4. Almost all the participants could answer the question

correctly (81 out of 83). These participants make up 97.59% of the participants who answered this question.

B4- "Not answered question" has been answered correctly by more than 97% of participants which is a high percentage indicating the ease of its recognition.

The results show that LAs of types A3 and B4 are very easy to recognise and more than 94% of participants could recognise them. These LAs' impact on program comprehension was insignificant which we believe is due to the ease of their recognition.

Conclusion for the second set of RQs

The results show that when the participants are educated about LAs their recognition rate is improved making them much easier to recognise after learning about them.

5.3 Conclusion for First and Second Sets of Research Questions

Table 5.9 shows the results from the first and second sets of research questions. The results show that A2 has a statistically significant impact with a small effect size because it is harder to recognise, with a 35.2% rate of correctly recognised occurrences before being taught about LAs. E1 also has a statistically significant impact but with a negligible effect size because it is easier to recognise, with a 63.63% recognition rate. After participants are educated about LAs, when all the recognition rates are very high, neither A3 (with a recognition rate of 94.8%) nor B4 (with a recognition rate of 97.59%) have a statistically significant impact on program comprehension. The results also show improvements in both program comprehension and LA recognition rates. Program comprehension improved as the significant impact of LA presence with small or negligible effect sizes in BeforeLecture reduced to insignificance impact in AfterLecture. LA recognition rate improved from a maximum of 63.63% to a minimum of 94.8%.

Summary of the conclusions of the first and second sets of RQs

We made the hypothesis H0 that LAs have a small impact on program comprehension because they can be easily recognised by participants. The results from our first and second sets of RQs support our hypothesis (H0) and show that LA presence does not have a substantial impact on program comprehension, their impact is at most small or negligible due to their ease of recognition by the participants. We thus conclude that linguistic antipatterns may not truly function as antipatterns considering their minimal impact on program comprehension. Also, we showed that learning about LAs can reduce their negative impact on program comprehension by improving their recognition rate..

Table 5.9: Resluts of First and Second Sets of RQs

LAs	**Impact**		**Recognition**
	Significance	Effect size	Correctness
BeforeLecture			
A2	Yes	Small	35.2%
E1	Yes	Negligible	63.63%
AfterLecture			
A3	No	∅	94.8%
B4	No	∅	97.59%

Chapter 6

Discussion

We now discuss further aspects of our experiments and their results to draw our final conclusion.

6.1 Linguistic Anti-patterns

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

D1- “Says one but contains many” We found a large number of real examples in our studied systems in which this type of LA occurs but we did not consider them as LA. We observed that developers prefer to choose simple names, like `tmp` for a stack of arrays, `x` and `y` for arrays of dimensions, or `v` for a collection of vectors. Here is an example of Apache-ant-1.10.1.

Listing 6.1: Example of an exception of “D1” (Apache-ant-1.10.1)

```
int size = LEFT_COLUMN_SIZE - label.length();
StringBuffer tmp = new
StringBuffer();
for (int i = 0; i < size; i++) {
    tmp.append("␣");
}
```

We decided not to consider such names as linguistic anti-patterns. Instead, they should be regarded as exceptions within this language convention. While such naming has the potential to reduce code readability, its impact might be mitigated, considering developers frequently employ such simple names.

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

Listing 6.2: Example of an exception of “E1” (JFreeChart 1.0.19)

```
// ** The maximum number of lines for category labels. */
private int maximumCategoryLabelLines;
```

Therefore, for numerical variables, plural names with singular type could be meaningful, and such naming should be considered as exceptions for this LA, since those LA appearances have a small impact on program comprehension, because of the meaningful association of variable names

with their intended meanings.

“Opposite meaning” We found a new type of LA, for example:

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

The statement uses the `err` attribute to print some error messages, yet the message is positive. We propose to create a new LA that could be described as “the method name and its parameters are contradictory”. This particular case is the most subjective LA even among the authors of this paper. We agree that some people might consider it an LA, while other people might argue that it is just for redirection and, therefore, not an LA.

6.2 Extended Analysis: Recognition Rates of Additional LAs (D1, F1, F2)

We studied the recognition rates of three other LAs, D1, F1, and F2, to understand whether the recognition rates found for the other four LAs are representative of other LAs. We assessed the recognition rates for these LAs in both experiments (BeforleLecture and AfterLEcture) to understand if there is an improvement in their recognition rates after giving participants a lecture on LAs. First, we discuss the results obtained from **BeforeLecture**.

D1- “Says one but contains many” Only participants in Group B from BeforeLecture answered questions on code snippets containing D1, in Q4.a. The results show that only one participant out of 110 participants (0.90%) could recognise this LA correctly. The other participants either did not answer the question or answered it wrongly.

D1- “Says one but contains many” is the hardest LA to recognise among all seven types of LAs based on the proportion of correct answers.

F1- "Attribute name and type are opposite" Participants from BeforeLecture answered a question on systems containing F1 (Q3). A total of 22.70% of participants (52 out of 229 participants) identified this type of LA as a bad coding practice.

F1- "Attribute name and type are opposite" is difficult to recognise according to the proportion of correct answers from participants.

F2- "Attribute signature and comments are opposite" In BeforeLecture, participants in both groups answered questions on code snippets containing instances of LA of type F2. Group A had Q4.a and Group B had Q4.c. In total, 29 participants from Group A and 22 participants from Group B could recognise F2 correctly. They constitute 22.27% of participants in the first experiment.

F2- "Attribute signature and comments are opposite" is easier than "D1" but is still hard to reconise.

The results show that the LAs D1, F1, and F2 are difficult LAs to recognise before participants learn about LAs.

In the following, we discuss the results obtained on the recognition rates of these three LAs from **AfterLecture**. We do the same steps as for BeforeLecture but focus on the data obtained from AfterLecture. We want to know whether having knowledge of LAs can improve the LA recognition rate by participants. In the following, we discuss each type of these three LAs in more detail:

D1- "Says one but contains many" All participants answered one Question (Q3) containing D1 in AfterLecture. We obtained 83 correct answers out of 177 responses: only 47.36% of participants provided the correct answers, even though they were taught about this LA before the experiment.

D1- "Says one but contains many" is the hardest LA to detect among the seven studied LAs. In BeforeLecture, only 1 out of the 110 participants successfully recognised this LA. Although the proportion of correct answers improved after the participants were taught about LA, it is still the hardest LA to recognise among our studied LAs.

F1- "Attribute name and type are opposite" In AfterLecture, participants answered the question Q4.b, which involved code containing F1. We obtained 88 correct answers out of 177 provided answers, for a percentage of 49.71%.

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

F2- "Attribute signature and comments are opposite" In AfterLecture, participants answered Q4.a, which involved a code snippet containing an LA of type F2. Out of 177 participants, 166 participants could find the correct answer. They constitute 93.78% of participants.

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

The results show low recognition rates for these three LAs (D1, F1, and F2) before learning about LAs, therefore, we conclude that future work should study their impact on program comprehension. The results also show an improvement in the recognition rates of these LAs after participants learn about LA. Figure 6.1 and Table 6.1 present all the obtained discussed results and compare the recognition rate of these LAs before and after having knowledge of LAs.

Table 6.1: The Correctness Percentage for Different LAs (BeforeLecture and AfterLecture)

Type	**Correctness_Before (%)**	**Correctness_After (%)**
D1	0.9%	47.36%
F1	22.7%	49.7%
F2	22.2%	93.7%

6.3 Mitigating Factors

We also investigated to what extent other factors, like fluency in the language in which the identifiers and comments are written, programming experience, gender, age, work experience, and education, could explain the participants' performance. We established a dataset containing information about these mitigating factors. We aggregated this data in sets for English proficiency (high

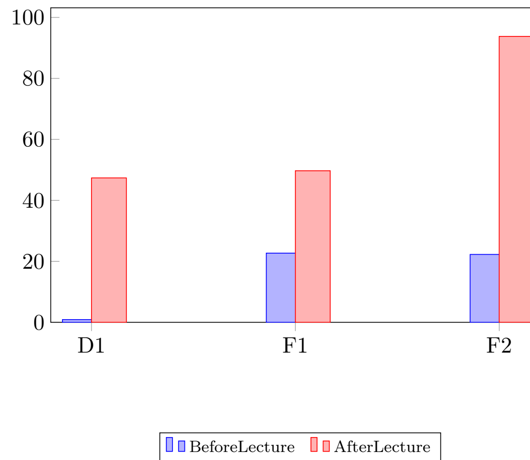


Fig. 1: LA Recognition Rates (Before and After Lecture

Figure 6.1: LA Recognition Rates (Before and After Lecture)

proficiency, low proficiency), programming knowledge (good, bad), age (18-25, 26-35, and 36-45), gender (male, female), degrees (Bachelors, Masters, PhDs, and others), and work experience (0 to 5 years, 5 to 10 years, and more than 10 years).

Given the variables' non-normal distribution and unequal sample sizes, the Mann-Whitney U test emerges as an appropriate statistical method for our analysis. Therefore, we utilized this test to assess the significance of the impact of our factors on program comprehension in terms of correctness, time, and effort. For factors consisting of data with more than two different ranges of values (such as age, work experience, and degree), we employed the Bonferroni correction method to control the familywise error which represents the probability of making at least one Type I error (false positive) when conducting multiple statistical tests simultaneously. The Bonferroni correction involves dividing our chosen significance level by the number of pairs of variables we tested against each other. After identifying statistically significant results, we calculated the average of the corresponding variables to determine the direction of these significant values. Furthermore, we computed the effect size to assess the practical significance of the statistically significant findings. All the results are available online[2].

6.3.1 Age, Gender and English proficiency level

We found no statistically significant differences between participants of different ages or from different genders or with different levels of English proficiency in terms of our dependent variables including correctness, time, and effort.

6.3.2 Programming knowledge

Correctness: We found a statistically significant result for Q3 in terms of correctness. Surprisingly, it contradicted our hypothesis, indicating that participants with greater programming knowledge provided fewer correct answers for Q3. However, the effect size is small. Q3 consists of D1 as the hardest (the most challenging) LA both before and after knowledge, which may confuse developers with a good knowledge of programming into choosing a wrong answer.

Time: There is a statistically significant finding for Q1 in terms of time, showing that participants having more programming knowledge spend less time answering the question. Q1 consists of A2 in BeforeLecture and A3 in AfterLecture, which are both easy LAs to recognise. Therefore, the easier the LA to recognise, the less time it takes to answer the question.

Effort: We found statistically significant results (with small effect sizes) for effort, showing that participants with more knowledge of programming put less effort into answering questions.

6.3.3 Degree

Correctness: We found a statistically significant result (with a small effect size) for Q3 in terms of correctness when comparing bachelors and others. The findings indicate that participants with a bachelor's degree exhibit a higher number of correct answers compared to other participants. We found the same result for Q3 by comparing participants in the Masters and Others groups. Question 3 includes D1, which is the hardest among others in both experiments, so it seems that participants, even after learning about LAs, have challenges finding the right answers when dealing with D1. The sets of masters and bachelors participants include Polytechnique students who have a good background in design patterns (as we discussed in 4.5) making them perform better when dealing with the most difficult LA.

Time: Considering the two sets of bachelor and others, and masters and others, there are statistically significant findings (with negligible effect sizes) for Q2 in terms of time. The results show that bachelor's students and master's students spend less time answering Q2 than others. Q2 consists of E1 in BeforeLecture and B4 in AfterLecture, which are both easy to recognise.

Effort: We found statistically significant (with small effect size) results for effort comparing bachelors and others, showing that bachelors put more effort into answering questions. The results are the same when comparing masters and others but with small and medium effect sizes. These results show that master's and bachelor's students are more precise about their answers.

6.3.4 Work experience

We found no statistically significant differences between participants with different work experiences in terms of correctness and time. When answering Q1, there is a statistically significant difference (with a medium effect size) between participants having 0 to 5 years of work experience and participants having more than 10 years of experience in terms of effort. Surprisingly, it contradicted our hypothesis by indicating that participants with more than 10 years of working experience make more effort to answer Q1 than participants with 0 to 5 years of experience.

6.4 Threats to Validity

Construct Validity: To assess construct validity, our study explored different aspects affecting participants' comprehension, including the correctness of their answers, time, and effort to evaluate their comprehehnsion. These measures are objective, but other factors, like fatigue, could change their values. We relied on the NASA TLX to measure the effort, which is inherently subjective because it is self-reporting. We accept the threat of self-reporting, and future work should consider other measures of the effort, possibly through eye-tracking or other more invasive means.

The NASA TLX used in our study underwent modifications from its original form, which involved reducing the number of scales. These adjustments may have implications for the construct validity of the NASA TLX scores. However, it's important to note that we excluded certain scales—including physical demands, temporal demands, and own performance because they were

not pertinent to our study. However, we recognize the potential impact of these modifications on the validity of our findings.

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

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

We used the PMD plugin to detect LAs by defining new rules related to each LA in Java systems. We accepted that the precision of this tool may not be 100%. Some false positive classes may have passed our manual validation if they "looked like" an LA. Conversely, our tool may have missed some true positive classes. Moreover, in addition to the impact of LAs, other characteristics in the code could have influenced participants' program comprehension. We accept these threats because we built the code snippets manually. We verified the code snippets and shared[2] them with the community for inspection and reuse.

We observed the positive impact of LA on time in the presence of LA for example in H1. This unexpected finding raises concerns about the accuracy of time recording or potential issues within the experiments themselves. We accept the threat related to potential issues in recording time.

We defined some keywords for each question and made a query based on these keywords to detect correct answers. The accuracy of our defined keywords affects our results because we use the number of correct answers to analyze our data. To mitigate this threat, we performed manual identification of correct answers for each question.

To evaluate the impact of different linguistic antipatterns (LAs), we used various LAs across

different experiments. The variation in LA types, along with the participants' prior knowledge in the second experiment, may have influenced the results, complicating the isolation of the impact of knowledge of LAs. This poses a potential threat to the validity of our work. However, this threat is almost mitigated, as the LAs selected for the study were the most negative LAs identified in previous work (Arnaoudova et al., 2016) and had similar complexity. This strategy for selection of LAs provided equally challenging questions in both experiments, allowing us to primarily observe the impact of knowledge of LAs for RQ2 as the main change in the second experiment.

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

Practical knowledge: It is possible that students had practical knowledge of linguistic antipatterns, yet were unaware of the academic definitions of LAs and their respective classifications, as this concept is a relatively recent development. Therefore, the lectures would have had little to no impact on their program comprehension of LAs. We accept the threats of having knowledge through practical experiences.

Learning with the experiment: Participants could learn about bad coding practices from BeforeLecture to perform better in AfterLecture, even if no training was conducted. We accept the threat of participants learning about LAs with experience.

Differential Participant Performance: Additionally, we noticed a notable difference in the performance of participants from Polytechnique University who showed faster and more accurate responses after learning about LAs. This distinction in performance may be because of their background in pattern and anti-pattern concepts based on the courses they have taken, which may have influenced their speed, resulting in more confidence and speed when answering questions with LAs.

Diffusion: Threats pertain to participants sharing tasks, questions, and answers. Although it is possible that a few participants exchanged some information, we defined two sets of questions to prevent too easy diffusion to prevent the participants with better knowledge of LA from sharing their answers and, thus, have more impact on our results. Therefore, this threat is almost mitigated, however, we accept the possible impact of diffusion threat on our results.

Selection Threats: Threats in the selection of participants concern the natural differences

among the participants' abilities, which could affect our results. We examined the possible impact of these confounding factors (using Age, Gender, English proficiency Level, Programming Knowledge, Work Experience, and Degree metrics). Results show that these factors did not significantly affect our dependent variables (i.e. correctness, time, and effort), which thus mitigates this threat. The additional table that shows the results of statistical tests is online[2].

Participant Variability: Some of the participants had prior knowledge of LAs and some of them did not complete the experiment process. However, we retained the data related to all the participants to have a larger sample size and achieve more precise results. The variability in participant characteristics posed a potential threat to the validity of our results. To mitigate this threat, we controlled the impact of these factors and noticed some differences in the average values of the dependent variables between participants with these factors and others; however, the differences were not substantial. To ensure the validity of our results, we also excluded those participants and analyzed the statistical results. We obtained the same conclusion that the presence of LAs has a small or negligible impact on program comprehension, and knowing LAs improves participants' program comprehension.

Conclusion Validity: Threat to conclusion validity concerns the correctness of the results. We asked participants to answer detection and comprehension questions about LAs in code snippets instead of reading the whole code and doing maintenance tasks. Participants could have answered the questions differently if they were asked to do the maintenance tasks and were given the whole code because they could have had a broader view of the system in hand. Due to the nature of experiments with participants where we wanted participants to only focus on LA in a piece of code, we accept these threats as did many previous works.

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

Reliability Threat: Threat to Reliability validity is about the possibility of replicating this study. We studied ten open-source systems with 229 participants but do not claim that these results are representative of all systems or developers. We provide all the necessary data online[2] to help other researchers replicate our work.

External Validity: External validity threat is related to the generalisability of the results. We performed our study on ten different, real systems belonging to different domains and with different sizes as shown in Table3.1. We wrote questions such that participants needed to focus on some code snippets from a whole system. We cannot claim that our results can be generalized to other systems written in other programming languages and to other participants. Future works should extend this study by considering other LAs, other participants, other questions, and other systems.

Chapter 7

Conclusion and Future Work

Linguistic Anti-patterns (LAs) are poor practices in the naming, documentation, and implementation of source-code entities, which are conjectured to affect the understanding of systems. In this study we made a hypothesis that LAs have a small impact on program comprehension because they can be easily recognised by developers. We performed two experiments (BeforeLecture and AfterLecture) to collect quantitative evidence about the impact of LAs on participants' comprehension of code snippets before and after a lecture on LAs. We also investigated the impact of having knowledge of LA on program comprehension and assessed the recognition rate of different types of LAs in our study to see their relation with program comprehension.

We considered seven different types of LAs for our study. We chose these LAs because they were considered to be poor practices by developers and they frequently occur in systems as shown by several previous works. We collected data before and after teaching participants about LAs. We established eight hypotheses to measure the impact of the presence of LA and knowledge of LA on program comprehension. To assess them, we used three metrics: (1) the amount of correct answers, (2) the time spent performing tasks, and (3) the NASA task load index for effort. We conducted statistical tests on these metrics to analyze the results and validate our hypotheses. We also investigated the recognition rates of these LAs because we believed LAs do not have an impact on program comprehension as they are easily recognisable by developers who are not thus impacted by their presence. We also assessed the impact of knowledge of LA on recognition rates. A total of 229 participants from two Universities participated in our experiments.

Results showed that linguistic anti-patterns (LAs) do have an impact on program comprehension but this impact is mitigated (small or negligible) when the LAs are recognised by developers. The results also show that learning about LAs can reduce their negative impact by increasing their recognition rate. It would be therefore beneficial to increase the awareness of LAs among students, as it can lead to improved program comprehension and correctness.

Thus, we conclude with the potential negative outcome that linguistic antipatterns may not effectively function as antipatterns, given their minimal impact on program comprehension.

Main conclusion

We conclude that the presence of LAs negatively affects program comprehension but with a small or negligible effect size due to their ease of recognition by participants. We also conclude that learning about LAs positively affects program comprehension by reducing the negative impact of LA and improving the recognition rate. Therefore, improving knowledge of LAs among students and developers is beneficial.

In future work, we want to study other types of LAs to see their impact on program comprehension. We also want to reproduce this experiment with other participants, in particular from the industry, to generalize our results. Finally, we want to study other programming languages, specifically dynamic programming languages, in which developers rely even more on names to compensate for the lack of types.

Appendix A

BeforeLecture-GroupA

Quality Code Activity 1

Group A

Please Enter Your Student ID: *

For this activity, you need to perform some tasks on two java classes and answer the questions. Besides, your task is to find coding practices on different code snippets, and finally write your feedback. We have designed this activity to be as short and simple as possible. It should not take more than 1 hours.

Next

First Question:

00:13

Question 1: Which method is used to determine the value of the "restricted" attribute in the following link (Click on the link to show the snippet code)? *

https://github.com/Resources-aks/Activity1-Group1/blob/master/RuntimeConfigurableWX.java

Back Next

https://github.com/Resources-aks/LA-Paper/blob/master/Activity1-Group1/RuntimeConfigurableWX.java

Second Question:

01:02

Question 2: At Line 912: does the assignment "=" affect a single or a collection of variables? in the following link? *

https://github.com/Resources-aks/Activity1-Group1/blob/master/DirectoryScannerWY.java

Back Next

https://github.com/Resources-aks/LA-Paper/blob/master/Activity1-Group1/DirectoryScannerWY.java

00:12

Question 3: Identify element (s) in the pieces of code that causes confusion and make understand-ability harder (click on the following link): *

https://github.com/Resources-aks/Activity1-Group1/blob/master/ActionNavigability.java

Back Next

https://github.com/Resources-aks/LA-Paper/blob/master/Activity1-Group1/ActionNavigability.java

Question 4: For each of the following five code snippets;

Identify bad coding practice (if any) in the code and-or comments and justify your answer.

Then, suggest ways to correct it (them)?

```
39 public final class SuffixLines
40     extends BaseParamFilterReader
41     implements ChainableReader {
42     /** Parameter name for the prefix. */
43     private static final String SUFFIX_KEY = "suffix";
44
45     /** The suffix to be used. */
46     private String suffix = null;
47
48     /** Data that must be read from, if not null. */
49     private String queuedData = null;
50
51     /**
52      * Constructor for "dummy" instances.
53      *
54      * @see BaseFilterReader#BaseFilterReader()
55      */
56     public SuffixLines() {
57         super();
58     }
59 //...
60 }
```

Please justify your answer: *

```
289     public void setCurrentProject(Project newValue) {
290         Project oldValue = currentProject;
291         currentProject = newValue;
292         firePropertyChange("currentProject", oldValue, newValue);
293     }
```

Please justify your answer: *

```
1552    protected boolean isSelected(final String name, final File file) {
1553        if (selectors != null) {
1554            for (int i = 0; i < selectors.length; i++) {
1555                if (!selectors[i].isSelected(basedir, name, file)) {
1556                    return false;
1557                }
1558            }
1559        }
1560        return true;
1561    }
```

Please justify your answer: *

Back Next

Feedback

Please select your age *

Please select your gender *

Please select your highest level of education completed *

- Bachelor's degree
- Master's degree
- Doctoral degree
- others

How much are you fluent in English? *

- bad
- neutral
- good
- excellent
- expert

How much work experience do you have? *

- < 3 years
- 3-5
- 5-10
- > 10

How do you rate your skills and knowledge in programming? *

- bad
- neutral
- good
- excellent
- expert

How do you rate your level in Java? *

- Never touched it
- Some notions
- I can code in java
- I have good skills in Java, I use it frequentl
- Expert

Back Next

How do you rate your level in Eclipse? *

- ☐ Never touched it
- ☐ Some notions
- ☐ I can code in java
- ☐ I have good skills in Java, I use it frequently
- ☐ Expert

Did you take the course LOG8430 (Architecture logicielle et conception avancée), and--or LOG8371(Ingénierie de la qualité en logiciel) before? *

Did you know what means Linguistic Anti pattern? *

Question 1

Mental Demand *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Effort *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Frustration *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Question 2

Mental Demand *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Effort *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Frustration *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Question 3

Mental Demand *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Effort *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Frustration *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Qustion 4

Mental Demand *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Effort *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Frustration *

	1	2	3	4	5	
Low	○	○	○	○	○	High

Any final comments?

Submit Form

Bibliography

V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *2013 17th European conference on software maintenance and reengineering*. IEEE, 2013, pp. 187–196.

W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE, 2011, pp. 181–190.

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

J. Bogner, R. Verdecchia, and I. Gerostathopoulos, "Characterizing technical debt and antipatterns in ai-based systems: A systematic mapping study," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2021, pp. 64–73.

F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are restful apis well-designed? detection of their linguistic (anti) patterns," in *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings 13*. Springer, 2015, pp. 171–187.

R. Brooks, "Towards a theory of the comprehension of computer programs," *International journal of man-machine studies*, vol. 18, no. 6, pp. 543–554, 1983.

S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 286–296. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196347

N. Borovits, I. Kumara, P. Krishnan, S. D. Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "Deepiac: deep learning-based linguistic anti-pattern detection in iac," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 7–12.

N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. D. Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel, "Findici: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code," *Empirical Software Engineering*, vol. 27, no. 7, p. 178, 2022.

L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B. C. Fung, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Investigating the relation between lexical smells and change-and fault-proneness: an empirical study," *Software Quality Journal*, pp. 1–30, 2015.

B. Caprile and P. Tonella, "Restructuring program identifier names." in *icsm*, 2000, pp. 97–107.

E. Merlo, I. McAdam, and R. De Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 4, pp. 205–244, 2003.

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

N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1998, p. 4.

P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, "A decade of code comment quality assessment: A systematic literature review," *Journal of Systems and Software*, vol. 195, p. 111515, 2023.

S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 95–99.

L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 145–158.

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

S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 260–269.

A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205–227, 2011.

S. L. Abebe and P. Tonella, "Automated identifier completion and replacement," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 263–272.

S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 602–611.

N. Al Madi, "Namesake: A checker of lexical similarity in identifier names," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

A. Blasi and A. Gorla, "Replicomment: identifying clones in code comments," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 320–323.

F. Palma, T. Olsson, A. Wingkvist, and J. Gonzalez-Huerta, "Assessing the linguistic quality of rest apis for iot applications," *Journal of Systems and Software*, vol. 191, p. 111369, 2022.

S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. Ieee, 2011, pp. 125–134.

S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can lexicon bad smells improve fault prediction?" in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 235–244.

E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 25–35.

F. Shull, J. Singer, and D. I. Sjøberg, *Guide to advanced empirical software engineering*. Springer, 2007.

K. Toutanvoa and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*, 2000, pp. 63–70.

J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

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

S. E. Dreyfus and H. L. Dreyfus, "A five-stage model of the mental activities involved in directed skill acquisition," Operations Research Center, University of California, Berkeley, Tech. Rep., 1980.