Titre: Title:	Multi-Language Design Smells: Characteristics, Prevalence, and Impact
Auteur: Author:	Mouna Abidi
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Abidi, M. (2021). Multi-Language Design Smells: Characteristics, Prevalence, and Impact [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/6304/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6304/
Directeurs de recherche: Advisors:	Foutse Khomh
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Multi-language Design Smells : Characteristics, Prevalence, and Impact

MOUNA ABIDI
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Mai 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Multi-language Design Smells : Characteristics, Prevalence, and Impact

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

Giuliano ANTONIOL, président
Foutse KHOMH, membre et directeur de recherche
Houari SAHRAOUI, membre
Denys POSHYVANYK, membre externe

DEDICATION

To my parents
To my sister
To my fiancé
To my friends
For their endless love, support, and encouragement

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and my deepest gratitude to my supervisor, Professor Foutse Khomh for leading me through the winding road of research, for encouraging and believing in me, for the economical and emotional support, and for sharing an invaluable amount of time to revise my work for the duration of my thesis.

Furthermore, I would d like to thank the members of my Ph.D. committee, Professor Giuliano Antoniol, Professor Houari Sahraoui, and Professor Denys Poshyvanyk, who enthusiastically accepted to review my dissertation.

I would like also to thank my colleagues and friends who collaborated with me during my thesis, some of them as co-authors in some papers, and some others as anonymous heroes, who were behind the scenes.

I would like to thank my beloved family for their unconditional support and endless love. You were my strengths through these difficult four years and without you, this day would never have existed and I would never get to achieve my dream. I love you and may God bless you all.

My sincere thanks go also to all the pattern community of PLoP conferences, for all the comments and suggestions that greatly helped defining and documenting the catalogue of multi-language design smells.

I am also thankful to all the anonymous participants who answered the surveys presented in this thesis.

Related Publications

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

1. **Articles in journals**

   (a) **Mouna Abidi**, Md Saidur Rahman, Moses Openja, Foutse Khomh. Are Multi-language Design Smells Fault-prone? An Empirical Study, Published in Transactions on Software Engineering and Methodology (TOSEM), 2020, ACM. This paper is presented in Chapter 9 and Chapter 8.

   (b) **Mouna Abidi**, Md Saidur Rahman, Moses Openja, Foutse Khomh. Design Smells in Multi-language Systems and Bug-proneness: A Survival Analysis, Submitted to Transactions on Software Engineering and Methodology (TOSEM), 2021, ACM. This paper is presented in Chapter 10.

   (c) **Mouna Abidi**, Manel Grichi, Foutse Khomh. Industrial Lookout: How do Developers Deal with Multi-language Systems?, Submitted to the Journal of Systems and Software (JSS), 2021, Elsevier. A part of this paper is presented in Chapter 6.

   (d) **Mouna Abidi**, Md Saidur Rahman, Moses Openja, Foutse Khomh. Multi-language Design Smells: A Backstage Perspective, accepted in principle for publication in Empirical Software Engineering (EMSE), 2021, Springer (Phase 2 of the MSR Registered Report). This paper is presented in Chapter 11.

   (e) **Mouna Abidi**, Foutse Khomh, Yann-Gaël Guéhéneuc. Design Smells for Multi-language systems, In press for publication in the Springer LNCS Transactions on Pattern Languages of Programming (TPLoP). A part of this paper is presented in Chapter 7.

   (f) **Mouna Abidi**, Manel Grichi, Foutse Khomh. Trends of Multi-language Systems: Contrasting Academic and Practice Usage. Submitted to the Information and Software Technology (IST), 2021, Elsevier. A part of this paper is presented in Chapter 5.

2. **Conference articles**

   (a) **Mouna Abidi**, Manel Grichi, Foutse Khomh. Behind The Scenes: Developers' Perception of Multi-language Practices, In the Proceedings of the 29th Annual

International Conference on Computer Science and Software Engineering (CASCON 19). **Best Student Paper Award**. A part of this paper is presented in Chapter 6.

(b) **Mouna Abidi**, Foutse Khomh, Yann-Gaël Guéhéneuc. Anti-patterns for Multi-language Systems, In the proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP 19). This paper is presented in Chapter 7.

(c) **Mouna Abidi**, Manel Grichi, Foutse Khomh, Yann-Gaël Guéhéneuc. Code Smells for Multi-language Systems, In the proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP 19). This paper is presented in Chapter 7.

(d) **Mouna Abidi**, Moses Openja, and Foutse Khomh. Multi-language Design Smells: A Backstage Perspective, In the proceedings of the 17th International Conference on Mining Software Repositories (MSR) 2020 (Phase 1 of the MSR Registered Report). This paper is presented in Chapter 11.

The following publications are not directly related to the material in this dissertation, but were produced in parallel to the research contained in this dissertation.

1. **Articles in journals**

   (a) Manel Grichi, **Mouna Abidi**, Fehmi Jaafar, Ellis E.Eghan, Bram Adams. On the Impact of Inter-language Dependencies in Multi-language Systems: Empirical Case Study on Java Native Interface Application (JNI)). In the proceedings of the 20th IEEE International Conference on Software quality, Reliability, and security (QRS 20), included in a special issue of IEEE Transactions on Reliability.

   (b) **Mouna Abidi**, and Foutse Khomh. Towards the Definition of Design Patterns and Design Smells for Multi-language Systems. In press for publication in the Springer LNCS Transactions on Pattern Languages of Programming (TPLoP).

2. **Conference articles**

   (a) **Mouna Abidi**, and Foutse Khomh. Towards the Definition of Patterns and Smells for Multi-language Systems. In the proceedings of the 25th European Conference on Pattern Languages of Programs (EuroPLoP 20).

   (b) Manel Grichi, **Mouna Abidi**, Yann-Gaël Guéhéneuc, Foutse Khomh. State of Practice of Java Native Interface. In the proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON 19).

RÉSUMÉ

De nos jours, il est courant de voir des équipes de développement combiner plusieurs langages de programmation lors du développement d'un nouveau logiciel. Les sites Web populaires tels que Netflix, Facebook, Instagram et d'autres logiciels modernes sont construits à l'aide de plusieurs langages de programmation. Les avantages en termes de coût de la réutilisation et les avantages qu'offre chaque langage de programmation sont les deux principales raisons qui incitent à utiliser plusieurs langages dans le même système. Les développeurs tirent souvent parti des atouts de plusieurs langages pour faire face aux défis de la création de logiciels complexes. Cependant, malgré les nombreux avantages qu'offrent les systèmes multi-langages, ils ne sont pas sans défis. En effet, la combinaison de langages de programmation avec différentes sémantiques et règles lexicales ajoute une complexité supplémentaire aux activités de maintenance et à la compréhension des systèmes multi-langages par rapport aux systèmes mono-langage, *i.e.,* des systèmes développés avec un seul langage de programmation.

La qualité est l'une des préoccupations les plus importantes des logiciels afin de réduire les coûts de maintenance et d'évolution. La qualité du logiciel dépend en partie de l'adoption de directives de bonne pratiques, de patrons de conception et de l'identification et correction de mauvaises pratiques, *i.e.,* des défauts de code et de conception. Ces directives représentent un moyen efficace d'améliorer la qualité des systèmes multi-langages. Elles capturent les bonnes pratiques à adopter et les mauvaises pratiques à éviter. Les patrons de conception décrivent de bonnes solutions aux problèmes de conception récurrents. Au contraire, les défauts de conception sont des symptômes de mauvais choix de conception. Ils représentent des violations des meilleures pratiques qui indiquent souvent la présence de problèmes plus importants. Cependant, pour les systèmes multi-langages, la littérature manque encore d'un ensemble établi de directives, patrons de conception et défauts de conception portant sur la manière de combiner différents langages de programmation afin de maximiser leurs avantages.

Par conséquent, dans cette thèse, nous visons à améliorer l'assurance qualité des systèmes multi-langages en recherchant s'il existe des défauts de conception spécifiques aux systèmes multi-language, leur prévalence et leurs impacts sur la qualité des logiciels. Pour cela, nous commençons par conduire deux études primaires pour capturer l'utilisation, et identifier les avantages et les défis des systèmes multi-langage. Nous avons d'abord mené une revue systématique de la littérature afin de collecter l'état de l'art des systèmes multi-langage, *i.e.,* leur utilisation, leurs défis et de l'information sur les differentes combinaisons existantes de langages de programmation. Pour avoir un aperçu de la perception qu'ont les

développeurs des systèmes multi-langage, nous avons mené, dans une seconde étude, un sondage auprès de développeurs expérimentés. À travers ce sondage, nous fournissons une perspective industrielle sur les systèmes multi-langage, leurs avantages et leurs défis du point de vue des développeurs. Nous avons choisi de mener une étude empirique en employant un sondage, car le développement et la maintenance sont des activités manuelles effectuées par les développeurs. Ainsi, l'évaluation des développeurs est importante.

Pour atteindre nos objectifs, dans une troisième étude, nous avons exploité toutes les sources d'informations auxquelles nous avons pu accéder (littérature, spécification des langages de prgrammation, blog des développeurs). Nous avons observé de bonnes et de mauvaises pratiques dans le code ainsi que des problèmes signalés dans la documentation des développeurs et les rapports de bogues. Nous avons codé et catalogué ces pratiques observées et rapportons nos résultats sous la forme d'un catalogue de défauts de conception. Ces défauts de conception reflètent des mauvaises décisions de conception et de codage lors de l'intéraction entre différents langages de programmation. Nous avons développé une approche pour détecter automatiquement les défaults de conception. Dans une quatrième étude, nous étudions la prévalence et l'évolution des défaults de conception sur des projets open source. Nous évaluons également leur impact sur la propension aux bogues. Nos résultats montrent que les défauts de conception sont prévalents dans les projets étudiés et persistent tout au long des versions des systèmes. Nous constatons que les fichiers de code contenant des occurrences de défauts de conception multi-langage peuvent souvent être plus associés à des bogues que les fichiers sans ces défauts, et que certains types de défauts de conception sont plus associés aux bogues que d'autres. Dans une cinquième étude, nous examinons le temps de survie des fichiers de code contenant des occurrences de défauts de conception multi-langage en comparant le temps jusqu'à l'apparition des bogues dans des fichiers impactés par les défauts de conception et les fichiers qui ne contiennent pas ces défauts. Dans la dernière étude, nous avons réalisé un sondage pour évaluer la prévalence perçue des défauts de conception contenus dans notre catalogue, leur impact et leur criticité. Nous pensons qu'un sondage est le meilleur moyen de d'étudier la prévalence et les impacts perçus de notre catalogue de défauts de conception, car les développeurs sont ceux qui interagissent quotidiennement avec des systèmes multi-langage et qui font face aux défis qu'ils posent. Nos résultats, montrent que les défauts de conception contenus dans notre catalogue sont perçus par les développeurs comme étant prévalent et ayant un impact négatif sur la qualité des logiciels.

Les résultats présentés dans cette thèse représentent une première étape importante vers la définition des patrons de conception et des défauts de conception spécifiques aux systèmes multi-langages. Nous pensons que nos recommandations et suggestions aideront les chercheurs et les développeurs à améliorer la qualité des systèmes multi-langages.

ABSTRACT

Nowadays, it is common to see software development teams combine multiple programming languages when developing a new software system. Popular websites such as Netflix, Facebook, Instagram, and other modern software systems are built using several programming languages and technologies. The cost benefits of reuse and advantages of each programming language are two main incentives for using multiple languages in the same system. Developers often leverage the strengths of multiple languages to cope with challenges of building complex software. By using programming languages that complement one another, performance, productivity, and agility may be improved. However, despite the numerous advantages of multi-language systems, they are not without some challenges. Indeed, combining programming languages with different semantics and lexical rules adds an extra complexity to the maintenance activities and comprehension of multi-language systems compared to mono-language systems, *i.e.,* systems developed with one programming language.

Software quality is one of the most important concerns for software systems to reduce maintenance, and evolution costs. Software quality partly depends on adopting guidelines, idioms, patterns, and avoiding design smells *i.e.,* code smells and anti-patterns. These guidelines are an effective means of improving the quality of multi-language systems. They capture good practices to adopt and bad practices to avoid. Design patterns describe good solutions to recurring design problems. On the contrary, design smells are symptoms of poor design and implementation choices. They represent violations of best practices that often indicate the presence of bigger problems. However, for multi-language systems, the literature is still lacking an established set of guidelines, design patterns, and design smells on how to combine different programming languages in order to maximise their benefits.

Hence, in this thesis, we aim to support the quality assurance of multi-language systems by investigating if there are design smells specific to multi-language systems, and analyzing their prevalence and impacts on software quality. For that, we start by conducting two pilot studies to capture the usage, benefits and challenges of multi-language systems. The first pilot study is a systematic literature review that aimed to understand what researchers have investigated about multi-language systems *i.e.,* their usage, their challenges, and existing combination of programming languages. To get an overview of the developers' perception of multi-language systems, we conducted, in a second study, a survey with professional developers. Through this survey, we provide an industrial perspective on multi-language systems, their benefits, and challenges based on the developers' experience. We chose to conduct an empirical study

using a survey because development and maintenance are manual activities performed by developers. Thus, the developers' evaluation is important.

To achieve our objectives, in a third study, we mined all possible sources of information we were able to access *i.e.,* (literature, language specification, developers' blog, bug reports, open source projects). We identified good and bad practices in the code as well as issues reported in the developers' documentation and bug reports. We encoded and cataloged these observed practices and report our findings in the form of a catalog of multi-language design smells. These design smells are poor design and coding decisions when bridging between different programming languages. We also developed an approach to automatically detect multi-language design smells. In a fourth study, we investigated the prevalence and evolution of multi-language design smells on open source projects. We also evaluate their impact on fault-proneness. Our results show that the design smells are prevalent in the studied projects and persist throughout the releases of the projects. We also found that files with multi-language design smells can often be more associated with bugs than files without these smells, and that specific smells are more correlated to fault-proneness than others. In a fifth study, we investigated the survival time of multi-language smelly files by comparing the time until the bug occurrence in smelly files and compared that time to the survival of non-smelly files. Our results show that multi-language smelly files experience bugs faster than files without those smells. In the last study, we performed a survey with professional developers to assess the perceived relevance of the specified multi-language design smells, as well as their perceived impact and severity. Results show that the studied design smells are perceived to be relevant and to have a negative impact on software quality.

The results presented in this thesis are an important first step towards the elaboration of design guidelines for multi-language systems. We believe that our recommendations and suggestions will help researchers and practitioners improve the quality of multi-language systems.

Keywords: Design smells, Anti-patterns, Code Smells, Multi-language Systems, Mining Software Repositories, Empirical Studies, Survey, Statistical Analysis, Software Quality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

FFI: Foreign Function Interface
JNI: Java Native Interface
JVM: Java Virtual Machine
EIC: Excessive Inter-language Communication
TMS: Too Much Scattering
TMC: Too Much Clustering
UMD: Unused Method Declaration
UMI: Unused Method Implementation
UP: Unused Parameters
ASRV: Assuming Safe Return Value
EO: Excessive Objects
NHE: Not Handling Exceptions
NCO: Not Caching Objects
NSL: Not Securing Libraries
HCL: Hard Coding Libraries
NURP: Not Using Relative Path
MMM: Memory Management Mismatch
LRA: Local References Abuse

CHAPTER 1 INTRODUCTION

"Quality is free, but only to those who are willing to pay heavily for it."
DeMarco and Lister

1.1 Research Context

Modern software systems are moving from the use of a single programming language towards the combination of multiple programming languages [1–5]. These systems are referred to as multi-language systems [6], *i.e.,* software systems that are developed with a combination of components written with at least two programming languages having diverse lexical, semantic, and syntactical programming rules. Capers Jones reported in his book published in 1998, that at least one third of the software application at that time were written using two programming languages. He estimated that 10% of the applications were written with three or more programming languages [7]. Kontogiannis argued that these percentages are becoming higher with the technological advances [2]. Popular websites such as Netflix, Facebook, Instagram, and other modern software systems are built using several programming languages and technologies [3]. A common approach to develop multi-language system is to write the source code in multiple languages to capture additional functionality and efficiency not available in a single language. For example, a mobile development team might combine Java, C/C++, JavaScript, SQL, and HTML5 to develop a fully-functional application. The core logic of the application might be written in Java, with some routines written in C/C++, and using some scripting languages or other domain specific languages to develop the user interface [8]. These systems usually rely on Foreign Function Interface (FFI) to call from one programming language routines or services written in another programming language. Java Native Interface (JNI) and Python/C are some types of FFI [9]. We will discuss these concepts in more detail in Chapter 2.

Multi-language development offers several advantages to software engineering [3, 4]. For example, it allows developers to reuse existing components and external libraries to reduce the development time and cost [4, 10, 11]. By combining programming languages, multi-language systems also allows to overcome the weaknesses that could be related to some programming languages. Multi-language development also allows to accommodate legacy code [10]. The cost benefits of reuse and the advantages of each programming language are increasingly powerful reasons behind the proliferation of multi-language systems. By

combining programming languages, developers' productivity and agility (*i.e.,* the ability to act rapidly) may be improved [3].

Multi-language development is considered necessary when a single programming language cannot offer the desired level of functionality or speed, interact properly with the database or the desired delivery platform, or meet end user expectations. In the case of Windows and macOS applications, a common situation is to write the core logic in C(++), then customise the implementation of the user interface depending on the operating system. It is possible to use C# for Windows, and Swift for macOS. This saves time because it prevents developers from writing the core logic twice and dealing with different sets of bugs in the different versions of the application[1].

However, despite the numerous advantages of multi-language systems presented above (*e.g.,* leverage the strengths of multiple programming languages, code reuse, accommodation of legacy code, gain in development cost and time, etc), these systems also introduce new and additional challenges [3, 12–15]. During the software life cycle, about 47% of the maintenance effort is spent on understanding activities while only 25% is spent on modification activities [4]. The inherent differences among the programming languages warrant multi-language expertise for the developers. Besides, the correct implementation of inter-language communication imposes additional complexities and compatibility challenges. Indeed, multi-language development increases the cognitive overhead of code [3, 16, 17]. Such difficulties unfortunately might lead to bugs that are hard to detect and debug. During 2013, famous web sites, *e.g.,* Business Insider, Huffington Post, and Salon were inaccessible, redirecting visitors to a Facebook error page. This was due to a bug related to the integration of components written in different programming languages. The bug was in JavaScript widgets embedded in Facebook and their interactions with Facebook's servers[2]. Another example is a bug reported early in 2018, which was due to the misuse of the guideline specification when using the Java Native Interface (JNI), to combine Java with C/C++ in `libguests`[3]. There were no checks for Java exceptions after all JNI calls that might throw them. Another example is the bug `JRUBY-6279` in JRuby, a Ruby compiler targeting the Java virtual machine (JVM). The compiler deferred the choice of the concrete method to be called at runtime to the Java class `Bootstrap`, which in turns delegated the choice of mathematical operations to another class in which developers forgot to implement the corresponding method. Consequently, performing a simple comparison with a float number would cause the JVM to

[1] `https://softwareengineering.stackexchange.com/questions/370135/why-are-multiple-programming-languages-used-in-the-development-of-one-product-or`

[2] `https://www.wired.com/2013/02/facebook-widget-snafu/`

[3] `https://bugzilla.redhat.com/show_bug.cgi?id=1536762`

crash. Several problems were also reported in JRuby mainly related to incompatibilities between languages and missing checks of return values and crashes related to the C language[4]. Hundreds of users of Microsoft Word also reported problems when using some Visual Basic macros. These problems are due to a mismatch between unique identifiers in C++ libraries and Visual Basic libraries[5].

As the number of languages increases, so does the challenges related to the development and maintenance of these systems [4, 12, 18–22]. As detailed in the examples above, the issues resulting from the combination of programming languages are mainly due to the mismatch and the incompatibilities between programming languages if not properly integrated. The challenge does not come necessary from the combination of programming languages but could result form the lack of means, *i.e.,* formal guidelines, design patterns, design smells, and tools that could support the quality assurance of multi-language systems.

1.2 Problem Statement

Several studies in the literature have investigated challenges related to multi-language systems [3, 12–15]. The majority of these studies report that program comprehension and system's complexity are the main challenges of the quality assurance of multi-language systems [3, 5, 16]. Software quality is partially achieved by adopting formal guidelines, design patterns, and avoiding design smells. In the last decade, empirical studies reported that design smells hinder software comprehensibility and may increase the bug- and change-proneness [23,24]. However, these studies on design smells primarily focus on mono-language projects and do not consider the smells related to the interaction between programming languages (*i.e.,* multi-language design smells). Kochhar *et al.* [3] claims that the use of several programming languages significantly increases bug proneness. Previous studies by Khomh [25] and Kochhar *et al.* [3] asserted that design patterns and design smells are present in multi-language systems and suggest that researchers study them thoroughly.

Design patterns [26] describe good solutions to recurring design problems. On the contrary, design smells describe poor solutions to design problems [27, 28]. Design smells reflect symptoms of poor design and implementation choices that may potentially have a negative impact on software quality. Developers working under a tight schedule, and-or who do not have adequate knowledge or experience required to solve a specific problem, often make poor decisions leading to design smells. Such poor decisions may lead to a variety of maintenance

[4] `https://www.jruby.org/2012/05/21/jruby-1-7-0-preview1.html`

[5] `https://support.microsoft.com/en-us/help/292744/bug-automation-client-receives-an-error-message-or-crashes-when-the-cl`

challenges and issues, including the increase of maintenance activities and the introduction of bugs [23, 24, 29–33]. While a design smell may not definitively identify an error, its presence suggests a potential trouble spot, a place where there is an increased risk of future bugs or potential failures.

However, despite the importance and increasing popularity of multi-language systems, the literature is still lacking an established set of guidelines *i.e.,* design smells and design patterns to follow or avoid when combining different programming languages in order to maximise their benefits [3,15,34]. The information of multi-language good and bad practices (e.g., [11,14,15]) is scattered between different resources [34]. Developers and researchers may not easily access these resources. This diversity of sources of information does not make it easy for developers to clearly identify which practices to adopt and which ones to avoid (to benefit from using multiple programming languages). This diversity also adds additional challenges to the development and maintenance of multi-language systems. Developers working on any part of the system are required to have experience in multiple programming languages. Moreover, they should consider the compatibility and rules *i.e.,* semantic, lexical, and syntactical related to each programming language to correctly handle the inter-language communications.

1.2.1 Thesis Hypothesis

The goal of our thesis is to improve the quality of multi-language systems. While multi-language systems offer opportunities for code reuse and the possibility to leverage the strengths of multiple programming languages, the heterogeneity of components written in different languages also impede code comprehension and increase maintenance overhead [1–3, 12]. As a result of this, developers may introduce design smells by making poor design and implementation choices. Therefore, We believe that being aware of the presence of occurrences of design smells in multi-language systems and understanding their impact on software quality may help developers and maintainers improve maintenance activities, and software quality. Therefore, in this thesis we formulate the following statement:

Thesis Statement:
Design smells (1) exist in multi-language systems; (2) they are prevalent in open source projects; and they (3) negatively impact the software quality.

1.2.2 Thesis Contributions

This thesis makes the following contributions:

1. A Systematic Literature Review (SLR) on the state-of-the-art of multi-language systems. (Presented in **Chapter 5**)
2. An investigation of developer's perspectives about the benefits and challenges of multi-language systems. (Presented in **Chapter 6**)
3. A catalogue of multi-language design smells. (Presented in **Chapter 7**)
4. An approach to automatically detect multi-language design smells in the context of JNI systems. (Presented in **Chapter 8**)
5. An analysis of the prevalence of multi-language design smells in open source projects. (Presented in **Chapter 9**)
6. A validation of multi-language design smells with professional developers (*i.e.,* relevance and impact). (Presented in **Chapter 11**)
7. An empirical evaluation of the impacts of multi-language design smells on software fault-proneness. (Presented in **Chapter 9** and **Chapter 10**)
8. An identification of activities that are more likely to introduce bugs once performed in files with design smells. (Presented in **Chapter 9**)
9. A categorization of bugs frequently occurring in multi-language smelly files. (Presented in **Chapter 10**)

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. **Chapter 2** provides background information, while **Chapter 3** surveys related work on multi-language systems and designs smells. **Chapter 4** provides a high-level overview of the research process followed in this dissertation. **Chapter 5** reports the state-of-art of multi-language systems and highlights the combinations of programming languages frequently used in multi-language systems. **Chapter 6** reports on the developers' perception of multi-language systems and their challenges. **Chapter 7** defines a catalogue of multi-language design smells, while **Chapter 8** describes an approach to detect multi-language design smells from our catalogue. **Chapter 9** examines the prevalence of multi-language design smells in open source projects and their impacts on software fault-proneness. **Chapter 10** reports on the survival of multi-language smelly files before the introduction of bugs, and provides categories of bugs that frequently occur in multi-language systems. **Chapter 11** reports about developers' perception of the relevance and impact of multi-language design smells. Finally, **Chapter 12** draws conclusions, discusses the limitations of our work, and outlines some avenues for future work.

CHAPTER 2 BACKGROUND

2.1 Chapter Overview

In this chapter, we present the necessary background materials related to multi-language systems, design patterns, design smells, and multi-language design smells. We also present and explain the statistical tests used in this thesis. This chapter is aimed for readers who are unfamiliar with these concepts.

2.2 Multi-language Systems

Modern software systems are moving from the usage of a single programming language towards the combination of several programming languages [1–3]. Most of the systems with which we interact daily integrate components written in several, different programming languages and technologies. Such systems are gaining popularity because of their different inherent benefits [18, 19, 35–40]. Developers often leverage the strengths of different programming languages to cope with the pressure and the challenges of building complex systems [18,19,35]. However these systems also introduce new and additional challenges related to their development and maintenance [1–3, 16].

Multi-language systems usually rely on the Foreign Function Interfaces (FFIs) [9] available in many programming languages to access, from one programming language, features and services available in libraries or components, written in other programming languages. Java Native Interface (Java/C(++)), Python C extensions (Python/C(++)), and Ruby C extensions (Ruby/C(++)) are some types of FFI. In this thesis, we study one specific type of FFI, the JNI systems (*i.e.,* systems with Java and C/C++ programming languages). For this reason, we present in the following an overview of the JNI development.

2.2.1 Java Native Interface Overview

While some applications could be entirely developed in Java, there are situations where Java alone does not fully meet the needs of the application. In such situations, developers use Java Native Interface (JNI) by combining Java and native code. JNI is a foreign function interface programming framework for multi-language systems. JNI enables developers to invoke native functions from Java code and also Java methods from native functions [41, 42]. JNI presents a simple method to combine Java applications with either native libraries and/or

applications [41, 42]. It allows Java developers to take advantage of specific features and functionalities provided by native code. JNI allows to perform hardware and platform-specific features. It also increases the performance with the help of low-level libraries for computation and graphic operations [11][1].

JNI Prerequisites In the following, we present fundamental concepts that could help to better understand the JNI development. More information about the JNI concepts are available in the literature (e.g., [41]).

1. **Java Elements:**
    - Keyword native, all methods that are defined in the Java code using the keyword native, must be implemented in a native shared library.
    - System.loadLibrary, a static method that loads a shared library to make the native functions implementations available to the Java code.
2. **C/C++ Elements:**
    - JNIEXPORT, used to mark a native implementation function as exportable into the shared library. This function will be then included in the function table, and thus JNI can easily locate it.
    - JNICALL, similar to JNIEXPORT, it ensures that the native methods are available through the JNI.
    - JNIEnv, a structure that contains the methods that can be used by the native code to access Java elements.
    - JavaVM, a structure that allows manipulating a running Java Virtual Machine.

JNI Illustration Example We present in Fig. 2.1 an example of a JNI code extracted from [41]. Figure (a) presents a Java class that contains a native method declaration `Print()` and loads the corresponding native library while Figure (b) presents the C file that contains the implementation of the native function `Print()`. `JNIEXPORT` and `JNICALL` are the macros needed to link the native method declaration in Java with its corresponding implementation in C. These macros ensure that functions are exported correctly through JNI [41].

[1] `https://hal.archives-ouvertes.fr/hal-01277940/document`

```java
/* Java */
class HelloWorld {
 static {
  AccessController.doPrivileged(
    new PrivilegedAction<Void>() {
   public Void run() {
    System.loadLibrary("HelloWorld");
     return null; }
          }   }
     private native void print();
     public static void
         main(String[] args) {
         new HelloWorld().print();
     }
 }
```

(a) JNI Method Declaration

```c
/* C */
 #include <jni.h>
 #include <stdio.h>
 #include "HelloWorld.h"


 JNIEXPORT void JNICALL
 Java_HelloWorld_print(JNIEnv *env,
    jobject obj)
 {

    printf("Hello World!\n");
    return;

 }
```

(b) JNI Implementation Function

Figure 2.1 JNI HelloWorld Example

2.3 Design Patterns and Design Smells

Design Patterns Patterns were introduced for the first time in the domain of architecture by Alexander [26]. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, In such a way that you can use this solution a million times over, without ever doing it the same way twice" [26]. From architecture, design patterns were introduced in software engineering by Gamma et al [43]. In their landmark book, Coplien et al. [44] provided an overview of practical guidelines for design pattern usage. They presented design patterns as a means to meet the goal of capturing the design of complex object-oriented systems. These design patterns are based on the developers' experiences when facing recurrent problems and applying "good" solutions to solve these problems. The goal of encoding and cataloging design patterns is to preserve, share, reuse, and improve design knowledge and take the benefits from similar, past situations [44, 45].

Design Smells Design smells are "opposite" to design patterns. They document "poor" solutions to recurring problems [27, 28]. "Poor", in this context, means that, for instance, the chosen solution can be ineffective, make the code unclear and difficult to maintain; increasing maintenance cost. They represent violations of best practices and often indicate the presence

of bigger problems [27,28]. Design smells include anti-patterns, which are higher-level design defects, and code smells, which are lower-level defects [27,28]. Code smells include low-level problems in source code, poor coding decisions that are symptoms of the presence of anti-patterns in the code. There exist several definitions in the literature for code smells and anti-patterns [27, 28, 46–48]. However, in this thesis we consider design smells, in general, to refer to both code smells and anti-patterns. Several studies in the literature studied the impacts of design smells in mono-language systems [23, 29, 32, 33]. They report that design smells affect software comprehensibility, increase change- and fault-proneness, and increase the effort needed to perform maintenance activities [29, 33]. For example, classes including smells are significantly more fault-prone and change-prone compared to classes without smells [23, 32].

Multi-language Design Smells Design patterns and design smells studied in the literature are mainly presented in the context of mono-language programming. While they were defined in the context of object oriented programming and mainly Java programming language, most of them could be applied to other programming languages. However, those variants consider mono-language programming and do not consider the interaction between programming languages. In a multi-language context, we define design smells as poor design and coding decisions when bridging between different programming languages. They may slow down the development process of multi-language systems or increase the risk of bugs or potential failures in the future [34, 49].

2.4 Statistical Techniques for Analysis

The analysis of the results of our experiments in this thesis involve classifiers, population tests, and correlation analyses. In the following, we discuss these statistical techniques.

2.4.1 Fisher's Exact Test

The Fisher's exact test [50] (used in Chapter 9) is a statistical test designed with the aim to assess if there are non-random associations between two categorical variables. The Fisher's exact test checks whether a proportion varies between two different samples. It involves testing the independence of rows and columns in a 2×2 contingency table based on the exact sampling distribution of the observed frequencies. This test is useful for categorical data that result from the classification of objects.

In our analysis, we also compute odds ratio (OR) [50] to complement the results of the

Fisher's exact test. The OR quantifies the strength of the association between two events of interest. It indicates the likelihood of a particular outcome (*e.g.,* an event occurrence). OR is calculated (as in Equation (2.1)) as the ratio of the odds p of an event occurring in a sample, *e.g.,* the odds that files with some specific design smells experience the event of interest *e.g.,* a bug (defined as experimental group), to the odds q of the same event occurring in another sample, *e.g.,* the odds that files with no smells experience a bug (defined as control group):

$$OR = \frac{p/(1-p)}{q/(1-q)} \tag{2.1}$$

An OR equal to 1 indicates that the event of interest is equally likely in both samples. While an OR greater than 1 means that the event is more likely to occur in the first sample (studied group). An OR less than 1 indicates that it is more likely to occur in the second sample (control group).

2.4.2 Correlation (Spearman's)

Spearman rank correlation (used in Chapter 9) is a non-parametric test that is used to measure the degree of association between two variables. This test is appropriate for both continuous and discrete ordinal variables. As in correlation analysis, in terms of the strength of relationship, the value of the correlation coefficient varies between +1 and -1. Therefore, the Spearman correlation between two variables of a value of $\pm$ 1 indicates that there is a perfect degree of association between the two variables. The relationship between the variables is weak when the correlation coefficient value goes towards 0. The direction of the relationship is indicated by the sign of the coefficient; a positive sign indicates a positive relationship and a negative sign indicates a negative relationship. For a sample size of n, if n ranks are distinct integers, the Spearman correlation coefficient can be computed as in Equation 2.2.

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2-1)} \tag{2.2}$$

Here,

- d_i=rg(X_i)-rg(Y_i) is the difference between the two ranks of each observation
- n is the number of observations.

2.4.3 Survival Analysis

Survival analysis (used in Chapter 10) are commonly used in medical research. Recently, researchers have been applying survival analysis to problems in Software Engineering [24, 51–

53]. Survival analysis are used to model the expected duration of time until the occurrence of one or more event(s) of interest. Several models are used to perform survival analysis. One of the most popular models for survival analysis is the Cox Hazard model, *i.e.,* Cox Proportional Hazards model. The purpose of Cox Hazard model is to evaluate simultaneously the effect of several factors on the survival of specific subjects under study. Cox Hazard model allows to capture how specific factors influence the rate of the occurrence of a well-defined event. The rate is defined as a hazard rate. The following function presents the hazard of the event of bug occurrence at a time t in Cox models:

$$\lambda_i(t) = \lambda_0(t) * e^{\beta * F_i(t)} \tag{2.3}$$

We obtain the following function when we take log from both sides:

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_1 * f_{i1}(t) + ... + \beta_n * f_{in}(t) \tag{2.4}$$

Here,

- $F_i(t)$ represents the function that defines the regression coefficient at the time t of the observation i.
- β corresponds to the coefficients that measure the impact of covariates in $\mathrm{F}_i(\mathrm{t})$.
- λ_0 is called the baseline hazard. It represents the value of the hazard if all the covariates are equal to zero.
- n represents the total number of covariates.

The baseline hazard λ_0 can be considered as the hazard of the occurrence of the event of interest (*e.g.,* bug occurrence) when no covariate presents an effect on that hazard. The baseline hazard would be omitted when formulating the relative hazard between two files at a specific time t, as shown in the following equation:

$$\lambda_i(t)/\lambda_j(t) = e^{\beta * (f_i(t) - f_j(t))} \tag{2.5}$$

Hazard ratios (HR) are measures of association widely used in prospective studies. It is the result of comparing the hazard function among the exposed group (*e.g.,* smelly files) to the hazard function among the non-exposed group (*e.g.,* non-smelly files). Hazard ratio can be considered as an estimation of possible risk, which is the risk of an event occurrence (*e.g.,* bug occurrence). A hazard ratio of 1 means that there is a lack of association, a hazard ratio greater than 1 suggests that there is an increased risk of the event occurrence, while a

hazard ratio below 1 suggests that there is a small risk that the event will occur. Therefore, usually the measures of association are presented along with their 95% confidence interval. This confidence interval allows to represent a reliable range of values in which we expect the true population parameter to be included instead of using a single value.

2.4.4 Machine Learning Techniques

In the following, we provide a short description of the machine learning algorithms used in this dissertation: logistic regression, and latent dirichlet allocation.

Logistic Regression

Logistic regression (used in Chapter 9) is a statistical model used to model the probabilities for classification problems with two possible outcomes. In a logistic regression model, the dependent variable is commonly a dichotomous variable that takes one of only two possible values when observed or measured, *i.e.,* 0 or 1. The multivariate logistic regression is based on the following Equation (2.6).

$$\pi(X_1, X_2, ..., X_n) = \frac{e^{\beta_0 + \beta_1 . X_1 + \ ... \ + \beta_n . X_n}}{1 + e^{\beta_0 + \beta_1 . X_1 + \ ... \ + \beta_n . X_n}} \tag{2.6}$$

Here,

- X_i are the independent variables for the logistic regression model, for example the number of smells of type S_i in a given source file and $S = \{S_1, S_2, ..., S_n\}$ is the set of the types of smells investigated.
- β_i are the model coefficients, and
- $0 \leq \pi \leq 1$ is the value on the logistic regression curve representing the probability of the occurrence of the studied phenomenon (*e.g.,* bug occurrence).

Each logistic regression model gives the log odds (regression coefficient estimate) of individual independent variables and their corresponding p-values. The log odds represent the factors by which the odds of the dependent variable will change for a unit change in values of corresponding independent variables. When the logistic regression coefficient is positive ($\beta_i > 0$), unit increase of the value of the corresponding independent variable will increase the log odds of the dependant variable by β_i assuming that other independent variables are either 0 or remain unchanged. For a negative regression coefficient ($\beta_i < 0$), on the other hand, the value of the log odds of the dependant variable will decrease by β_i for unit increase in the

value of the associated independent variable. Thus, the higher the positive log odds of an independent variable, the higher is the impact of that independent variable on the studied phenomenon (*e.g.,* bug occurrence).

Latent Dirichlet Allocation

Latent Dirichlet Allocation LDA (used in Chapter 9 and Chapter 10) is a generative statistical model for collections of discrete data such as text. LDA is a widely adopted topic modeling technique which is able to extract topics from small documents (*e.g.,* commit messages) [54]. LDA was used in several studies in the literature [55–59]. LDA generates topics based on a set of frequently co-occurring keywords. It treats the input data as a corpus of textual documents, that is used as a basis for topic modeling. Given a corpus of n documents f_1, ..., f_n, topic modeling techniques automatically discover a set Z of topics, $Z = z_1$, ..., z_k. The variable k presents the number of topics. It is an input that controls the granularity of the topics. LDA is a probabilistic approach that categorizes the topics after a set of iterations I. A document is composed by a vector of topic probabilities while a topic is presented as a vector of word probabilities. A topic that exhibits the highest proportional value is considered as the most dominant topic within that dataset.

2.5 Chapter Summary

In this chapter, we presented some background information about multi-language systems, we introduced design smells, and briefly discussed the statistical tests used in this dissertation.

CHAPTER 3 LITERATURE REVIEW

3.1 Chapter Overview

In this chapter, we survey the most relevant research work related to this dissertation, which includes multi-language systems, definition of design smells, their detection, and impacts on software quality attributes.

3.2 Multi-language Systems

Several studies in the literature discussed multi-language systems. One of the very first studies, if not the first, was by Linos *et al.* [60]. They presented *PolyCARE*, a tool that facilitates the comprehension and re-engineering of complex multi-language systems. *PolyCARE* seems to be the first tool with an explicit focus on multi-language systems. They reported that the combination of programming languages and paradigms increases the complexity of program comprehension.

Kullbach *et al.* [4] also studied program comprehension for multi-language systems. They claimed that program understanding for multi-language systems presents an essential activity during software maintenance and that it provides a large potential for improving the efficiency of software development and maintenance activities. They presented an approach to support the understandability of multi-language systems. This approach provides a graph-based conceptual modeling, in which models that represent relevant aspects of each programming language are built and integrated into a common conceptual model that combines between several programming languages.

Linos *et al.* [1] later argued that no attention has been paid to the issue of measuring the impact of combining multiple programming languages on program comprehension and maintenance. They proposed *Multi-language Tool (MT)*; a tool for understanding and managing multi-language programming dependencies. They studied MLPD (MuLti paradigmatic Program Dependencies), these dependencies arise when entities from one language interact with entities developed in another language. They studied the interaction between Java and C/C++. The tool is animated with gravity animation and displays circles for each programming language and their dependencies. The size of the circles is related to the number of lines of code. The tool is based on lexical analysis of keywords related to the call of another language's functions. They conducted a case study to evaluate their tool. They found that using MT, participants made fewer changes to the code than participants without access to

this tool, which may improve the efficiency of larger multi-language systems.

Kontogiannis *et al.* [2] stimulated discussion around key issues related to the comprehension, reengineering, and maintenance of multi-language systems. They performed discussion sessions to attract researchers with an interest in understanding and maintaining multi-language systems. They argue that creating dedicated multi-language systems methods and tools to support such systems is expected to automate and ease the comprehension and maintenance of multi-language systems.

Similarly, Kochhar *et al.* [3] studied 628 projects from GitHub to understand the impact on software quality of using several programming languages. They reported that the use of more than one programming language significantly increases bug proneness. They reported that design patterns and anti-patterns were present in their studied multi-language systems and suggested that researchers study them thoroughly. They also suggested that the community conducts further studies to investigate the benefit of using multiple programming languages in a system.

Other researchers carried out studies specific to JNI multi-language systems. Tan *et al.* [11] studied JNI usages in the source code of the JDK v1.6. They examined a range of bug patterns in the native code and identified six bugs related to the use of JNI methods in the JDK. To retrieve patterns, they collected empirical evidence, and characterised relevant bugs in JNI. They also relied on a list of warnings produced by static analysis tools that they used to identify the bugs. The identified bugs can lead to a crash in the JVM and can introduce vulnerabilities and expose the JVM to security breaches. They argued that bugs are likely to occur in JNI systems because of language incompatibilities and the assumptions made by the Java code regarding the C/C++ code (*e.g.,* buffer overflows, unexpected control flow path). They proposed static and dynamic algorithms to prevent these bugs. However, they limited their study to a portion of the native code in JDK v1.6 *i.e.,* java.util.zip in the Sun's JDK, about 800k lines of code.

Li and Tan [61], on the other hand, highlighted the risks caused by the exception mechanisms in Java, which can lead to failures in JNI implementation functions and affect security. They studied the bugs caused by a lack of management of the exceptions. They argued that such issues negatively impact the security of the system and introduce failures. They proposed a static analysis tool to examine and report potential risks in JNI systems. They focused mainly on JNI but argued that their approach could be adapted to other FFIs, such as Python/C and OCaml/C APIs. They also proposed [62] a static framework, JET, which extends Java's exception checking mechanism to cover native code to prevent failures and ease debugging.

Kondoh *et al.* [63] presented four kinds of common JNI mistakes made by developers. They proposed *BEAM*, a static-analysis tool, that uses a typestate analysis, to find bad coding practice pertaining to error checking, virtual machine resources, invalid references, and JNI methods in critical code sections. The authors reported that these mistakes may lead to several problems and lead to security vulnerabilities. They evaluated their tool and found that *BEAM* increased the total number of bug reports by 76%.

Ebert *et al.* proposed GUPRO an approach to support the understanding of multi-language code. The approach relies on a graph-based conceptual model and focuses on the analysis of multi-language systems. It is built upon the entity-relationship dialect with a combination of the GRAL that is used for architectural constraint validation [64].

Synytskyy *et al.* Island Grammars [65] proposed Island Grammars for multi-language code analysis and reverse engineering which can be used in a situation where the parse tree is not complete. The approach goes through the source code and assigns each part to two possible classifications. Islands, to which it assign a detailed production for the code of interest and Water that refers to the remaining parts.

Ayers *et al.* [66] collected and analyzed bugs in multi-language systems. They proposed TraceBack, a tool that collects the bugs and stores them through run-time instrumentation of control-flow blocks. The approach uses an intermediate language representation to provide a unified trace of components' execution.

Strein *et al.* studied multi-language code analysis techniques and proposed an approach based on a language-independent meta-model [67]. This approach provides abstract representations and assigns them to the respective programming language or paradigm. They also proposed multi-language refactoring techniques through the support of generic language descriptions.

Linos *et al.* proposed a metric based approach related to multi-language systems build on top of Microsoft Intermediate Language (MSIL) [68]. The approach was proved to be effective for .Net platform. Bissyandé *et al.* [69] studied the popularity, interoperability, and impact of programming languages by analyzing a large number of open-source systems hosted on GitHub. They identified popular languages that are used to implement many projects, programming languages that are often combined, and correlations between individual language with systems' successes, numbers of issues, and team sizes. They studied the impacts of programming languages on the software quality. They reported that multi-language development should be used with caution. They concluded that earlier popular languages, such as the C programming language, are still frequently used with a large code base, while the rush for web development has made JavaScript and Ruby prevalent. They also found that ObjectiveC, the language mainly used in *Apple* product has also been gaining much attention.

Wampler *et al.* citewampler2010guest discussed the trends of multi-language programming. One of these trends is the use of different programming languages executing in the same virtual machine .They presented the advantages of dealing with requirements of today and future software systems. They claimed that Erlang presents the best prototype of the successful use of FP and actors in industrial use to implement robust telecom switches with many concurrent processes. It has been used in project such as GitHub, Riak, etc. They reported that the use of multiple languages executing on the same Virtual machine positively impacts the interoperability between components written in different languages. They also highlighted that using kernel languages like C/C++ with scriptural language like Lua, Ruby provides more flexible and performing applications.

Gong *et al.* [70] analyzed the source code of JDK v1.2. They outlined the main limitations of its security features including, but not limited to, access rights, library loading, and exception management. They discussed issues raised by the deployment of Java technology. They also presented the new classes and protection mechanisms introduced with v1.2 and their usages. The authors discussed the impact of deploying the new features on existing code with respect to security. Unfortunately, they limited their study to one version and security features. Moreover, another limitation of this study is that the paper mostly introduced JDK and provided a whole overview of the packages and classes without performing code analysis.

Mushtaq *et al.* [16] performed a literature review for multi-language source code analysis. They highlighted the importance of addressing the complexity that results with modern multi-language systems. They presented the advantages and limitations of the few existing source code analyzes tools. For example, they argued that Multi-Language Metrics Tool is not extensible and supports only .NET environment languages. They reported that KDM (Knowledge Discovery Meta model) can be considered for multi-language analysis but it needs a specific model that can be extended to include multi-language analysis.

Mayer *et al.* [5] proposed an approach to understand, analyze, and refactor multi-language source code by explicitly specifying the semantic links between entities written in different programming languages. By semantic link they considered semantic connections used to link two or more artifacts written in different programming languages. Their approach includes the detection of build errors and failures due to missing links. The proposed approach reduces some challenges related to program understanding, code analysis, and refactoring when dealing with different languages but it focuses only on semantic links between languages and do not cover other kinds of challenges (*e.g.,* dependencies related to the development and maintenance) reported by developers when dealing with multi-language systems. Later, Mayer *et al.* [71] performed a survey to investigate multi-language development. Developers

reported that multi-language systems introduce benefits and challenges. However, in this study, they focused on the identifiers and cross-language links between languages and did not cover the challenges related to the development and combination of programming languages. This study also targeted Germany developers mainly from the same company. However, some of the questions were targeting concrete software projects. Thus, respondents in this survey may have reported about their experiences with these specific projects.

Moise and Wong [72] extracted and studied multi-language systems dependencies among heterogeneous components. Their approach use predefined extractors from *Source Navigator (SN)* to produce the facts inside each language. They used, for each programming language, available APIs provided by the tool SN, to identify multi-language calls. They studied the calls to JNI API between Java and C. They started by extracting the source code into models conform to the GXL format (Graphical eXchange Language). This approach is useful to retrieve the dependencies among multi-language systems. They evaluated their approach with two case studies to access its correctness and scalability. Their approach was used to detect defects that exist in the *Java-GNOME* system that may appear as run-time errors and that may not be detected without a cross-language analysis.

Fjeldberg [73] published a master thesis discussing multi-language programming. He defined multi-language systems as systems using more than one programming language within the same context. By context he considered the same or several teams that interact together in the same project. Where integrating each of the completed modules in a specific language require knowledge on all the languages used to complete the project. He conducted two cases studies with developers. He presented the advantages and disadvantages of multi-language programming: productivity and maintainability were the perceived advantages by participants and the perceived disadvantages were knowledge, maintainability, and tool support.

3.3 Design Patterns and Design Smells

Since their popularization by Gamma *et al.* [44], several studies have shown great interest for design patterns and design smells from their definition [74,75] to their detection, and impacts [76–78]. In the following we present related work on design smells definition, detection, and impacts analysis on software quality.

3.3.1 Definition of Design Patterns and Design Smells

Gamma *et al.* [79] introduced the first book on design patterns. They defined a design pattern as a solution to a recurrent design problem. Alexander [80] defined a pattern as a

three dimension rule which presents the context, the problem, and the proposed solution. He presented the patterns as a relationship between the recurrent problem to a specific context and the proposed solution that solves the problem.

Webster [48] introduced the first book of anti-patterns for object oriented programming. The book described the anti-patterns as a frequently used solution that did not provide the expected effect. Riel *et al.* later, proposed 61 heuristics describing object-oriented programming practices that could be used to assess a program's quality manually and improve its design and implementation. These heuristics are similar to code smells. Fowler *et al.* [28] defined 22 code smells. Brown *et al.* [27] described 40 types of anti-patterns. The book discussed the low level design smells, code smells as the basis of anti-pattern detection.

A few studies investigated the design patterns and practices related to multi-language systems [11, 13–15, 81, 82].

Neitsch *et al.* [15] performed a qualitative study on five multi-language systems. They identified issues related to the deployment of multi-language systems and specified some common build patterns and anti-patterns for multi-language systems that summarise the key problems related to the builds of multi-language systems, e.g., *Filename Collision, Installation Required, Unverified Third Party Software, Ignored Error, Incorrect Dependencies, Build-Free Extensibility, Object-Oriented Builds*. They found that many build problems can be systematically addressed.

Goedicke *et al.* [14] proposed five architectural patterns based on well-known design patterns. These patterns are defined to wrap legacy components as black-box entities. Most of the defined patterns can be used with different programming languages. To assess the legacy migration and the wrapping techniques, the authors also presented a pilot project. They also provided a detailed definition of these patterns. The pattern *Object System Layer* provides a highly flexible object system as a layer build on top of a given language [13]. It makes components that are not object-oriented or that are implemented in another language, accessible through *Object System Layer*. These components can then be treated as black-boxes. The pattern *Message Redirector* ensures a simple indirection architecture that maps the calls to a message implementation [81]. It also provides callback methods around the calls.

Malinova [83] made an attempt to connect some well-known design patterns e.g., Adapter, Proxy, and Wrapper Facade, to the process of Java wrapping of native legacy codes. In this paper, design patterns were studied in the context of invoking native applications from Java code.

Osmani *et al.* [84] presented the Lazy Initialisation pattern which describes how to execute

Ajax requests in JavaScript, where the Ajax request includes a URL and some data, possibly in JSON or XML, to communicate with a server, likely implemented in C/C++.

3.3.2 Design Smells Detection Approaches

Van Emden *et al.* [85] proposed the JCosmo tool that supports the visualization of the code layout and design smells locations. They used primitives and rules to detect occurrences of design smells while parsing the source code into an abstract model.

Marinescu *et al.* [86] proposed an approach for design smells detection based on detection strategies. The approach captures deviations from good design principles and heuristics to help developers and maintainers in the detection of design problems.

Lanza *et al.* [87] presented the platform iPlasma for software modeling and analysis of object oriented software systems to detect occurrences of design smells. The platform applies rules based on metrics from C++ or Java code.

Moha *et al.* [88] proposed a language based on a BNF grammar and a framework that supports the automatic generation of detection algorithms. The language describes the design smells metrics, structural, and semantics properties. The proposed language is easy to understand and modify. In addition, it is more flexible than implementing ad-hoc detection algorithms, because it describes design smells from a higher level of abstraction. Later, Moha *et al.* [89] introduced DECOR which detects design smells in Java programs. DECOR is based on a domain-specific language that generates the design defect detection algorithms.

Khomh *et al.* [90] proposed a Bayesian approach to detect occurrences of design smells by converting the detection rules of DECOR into a probabilistic model. Their proposed approach has two main benefits over DECOR: (i) it can work with missing data and (ii) it can be tuned with analysts' knowledge. Later on, they extended this Bayesian approach as BDTEX [91], a Goal Question Metric (GQM) based approach to build Bayesian Belief Networks (BBNs) from the definitions of anti-patterns. They assessed the performance of BDTEX on two open-source systems and found that it generally outperforms DECOR when detecting Blob, Functional Decomposition, and Spaghetti code anti-patterns.

Ouni *et al.* [92] proposed an approach to detect and refactor design smells occurrences. They used genetic programming to extract design smells detection rules from instances of maintainability defects. They used the generated detection rules to correct the design smells. The correction of design smells was the result of a combination of refactoring operations.

Kessentini *et al.* [93] proposed an approach based on genetic programming that relies on example of occurrences of design smells to generate the detection rules. The proposed approach

automatically finds detection rules and proposes correction solutions in term of combinations of refactoring operations. They reported that this approach presents some limitations as the generated rules depends on the coverage of the different behaviors of design smells. In another study, Kessentini *et al.* [94] proposed a detection approach based on the assumption that what significantly diverges from good design practices is likely to reflect a design smell. They evaluated the proposed approach and reported a precision of 95%.

Palomba *et al.* [95] proposed an approach to detect design smells using historical information. The proposed approach exploits change history information extracted from versioning systems to detect instance of five design smells. They evaluated the proposed approach and reported a precision between 72% and 86%, and a recall between 58% and 100%. They performed a second evaluation with professional developers and reported that over 75% of smell instances identified by the proposed approach were also recognized by developers as a design or implementation problem.

Rasool *et al.* [96] proposed an approach to detect occurrences of code smells that supports multiple programming languages. They argued that most of the existing detection techniques for code smells focused only on Java language and that the detection of code smells considering other programming languages is still limited. They used SQL queries and regular expressions to detect code smells occurrences from Java and C# programming languages. In their approach, the user should have knowledge about the internal architecture of the database model to use the SQL queries and regular expressions. In addition, each language needs a specific regular expression.

Fontana *et al.* [97] conducted a study applying machine learning techniques for smell detection. They empirically created a benchmark for 16 machine learning algorithms to detect four types of code smells. The analysis was performed on 74 projects belonging to the `Qualitas Corpus` dataset. They found that `J48` and `Random Forest` classifiers attain the highest accuracy (>96%).

Liu *et al.* [98] proposed a smell detection approach based on Deep Learning to detect Feature Envy. The proposed approach relies on textual features and code metrics. It relies on deep neural networks to extract textual features.

Barbez *et al.* [99] introduced a deep-learning approach that relies on structural and historical information of code metrics to detect design smells occurrences. The authors evaluated the proposed approach by detecting occurrences of the design smell God Class on three open source projects. They reported that the use of historical values of code metrics improves the performance of the proposed approach. Barbez *et al.* [100] proposed a machine learning based method SMAD that combines several code smells detection approaches based on their

detection rules. The core of their approach is to extract metrics based on existing approaches and use those metrics as features to train the classifier for smell detection. The proposed approach supports the detection of the smells of type God Class and Feature envy. Their approach outperforms other existing methods in terms of recall and Matthews Correlation Coefficient (MCC).

Palomba *et al.* [101] proposed TACO, an approach that relies on textual information to detect code smells at different levels of granularity. They evaluated their approach on ten open source projects and found that the proposed approach outperforms existing approaches.

Borrelli *et al.* [102] recently documented seven types of smells for video games. They proposed UnityLinter, a static analysis tool that detects occurrences of video games design smells. They also surveyed 68 practitioners. They reported that developers are concerned by performance and behavior issues. However, they were less concerned by maintainability issues.

3.3.3 Design Smells Impact Analysis

Several studies in the literature have studied the impact of design smells on software quality but mainly for mono-language systems.

Khomh *et al.* [23] analyzed nine releases of Azureus and 13 releases of Eclipse to investigate if the classes with occurrences of design smells are more change-prone than classes without those occurrences. They concluded that the classes with occurrences of design smells are more likely to be the subject of changes than classes without those occurrences. In another study, Khomh *et al.* [31] investigated the impact of 13 design smells in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino. They reported that classes with design smells are more change- and fault-prone compared to classes without design smells. Our study presented in Chapter 9 is closely inspired from this study.

Olbrich *et al.* [103] proposed an approach that analyzes the evolution of design smells and study their impact on the frequency and size of changes. They study two design smells: God Class and Shotgun Surgery. They used an automated approach based on detection strategies to detect the occurrences of design smells. They identified different phases in the cycle of design smells evolution during the different phases of the system development. They also found that components infected by design smells exhibit different behavior.

Abbes *et al.* [104] investigated the impact of occurrences of anti-patterns on the developers' understandability of systems while performing comprehension and maintenance tasks. They conducted three experiments to collect data about the performance of developers and study the impact of Blob and Spaghetti Code anti-patterns and their combinations. They concluded

that the occurrence of one anti-pattern does not significantly impacts comprehension while the combination of the two anti-patterns negatively impacts program comprehension. This finding was corroborated by Politowski *et al.* [105].

Linares *et al.* [106] studied the potential relationship between the occurrence of design smells and quality attributes as well as the possible relation between design smells and application domains. They analyzed 1,343 Java Mobile applications in 13 different application domains. They concluded that anti-patterns negatively impact software metrics in Java Mobile applications, in particular, fault-proneness. They observed that there is a difference in the metric values between classes containing occurrences of smells and classes without smells. They also found that some smells are more frequently present in certain application domains.

Soh *et al.* [33] performed a study with six developers, three maintenance tasks, and four equivalent functions in Java. They used the Eclipse Mimec plugin and Thinkaloud sessions to analyze the effort spent by different developers when performing different maintenance activities (editing, reading, navigating, searching, static navigation, executing, and other activities). They concluded that design smells have different impacts on the effort needed to perform the different activities. They also found that the effort needed for reading, navigating, and editing is affected by three smells: "Feature Envy", "God Class", and "ISP Violation".

Saboury *et al.* [24] conducted a survival analysis of JavaScript design smells and compared the time to fault between files with and without JavaScript smells. They reported that JavaScript smells negatively impact the quality of JavaScript projects. We reused the methodology proposed by Saboury *et al.* to conduct the study presented in Chapter 10.

Muse *et al.* [107] performed an empirical study on the prevalence and impact of SQL design smells. They reported that SQL smells are prevalent and persist in the studied open-source projects and that they have a weak association with bugs.

Tufano *et al.* [108,109] performed an empirical study on the change history of 200 open source projects and investigated when and why design smells are introduced. They also studied the survivability of design smells and how they are removed by developers. They suggest that the design smells instances are introduced when an artifact is created and not as a result of maintenance and evolution activities. The results report that 80% of design smells once introduced are not removed by developers, while the 20% of the removed design smells are rarely resulting from refactoring.

Yamashita *et al.* [29] conducted a survey aimed at investigating developers knowledge about design smells and their perceived criticality. They surveyed a total of 85 professional devel-

opers and reported that 32% of the participants stated that they did not know about design smells. They reported that the perceived criticality differs from one type of design smell to the other, and that the majority of the participants were moderately concerned by design smells. We reused the demographic questions of this study to conduct the surveys presented in Chapter 6 and Chapter 11.

Palomba *et al.* [110] conducted a survey aimed at providing evidence on developers' perception of design smells. They surveyed master's students, general developers, but also original developers that contributed on the smelly files. They reported that some design smells instances are generally not perceived by developers as containing a design or implementation problem. Their results also suggest that developers' experience plays an important role in the identification of design smells instances. Our study presented in Chapter 11 is closely inspired from this study but applied in the context of multi-language systems.

In a recent study, kermansaravi *et al.* [111] investigated the mutation between design patterns and design smells. They reported that design patterns and design smells mutate into other types of design patterns and design smells during software evolution. They also highlighted that some mutations of design patterns and design smells are more likely to increase the risk of bugs.

3.4 Chapter Summary

In this chapter, we briefly discussed the literature on multi-language systems. We also presented the related literature about design smells, along with their definition, detection, and impacts on software quality. As presented in this chapter, several studies in the literature discuss multi-language systems. However, only a few of them investigate the good or bad practices that could be used to improve their quality. Therefore, there are no formal guidelines that developers' could consider when developing multi-language systems. The information about the practices is scattered between different resources and their impacts are still not yet evaluated in practice, *e.g.,* open source projects and developers' perceptions.

CHAPTER 4 RESEARCH OBJECTIVES AND ORGANIZATION OF THE THESIS

4.1 Chapter Overview

We present in this chapter an overview of the objectives and sub-objectives of the thesis. We also present the structure of this dissertation. As previously mentioned in Chapter 1, this thesis aims to define multi-language design smells and study their prevalence and impacts on software quality. Figure 4.1 presents an overview of the research process of this dissertation.

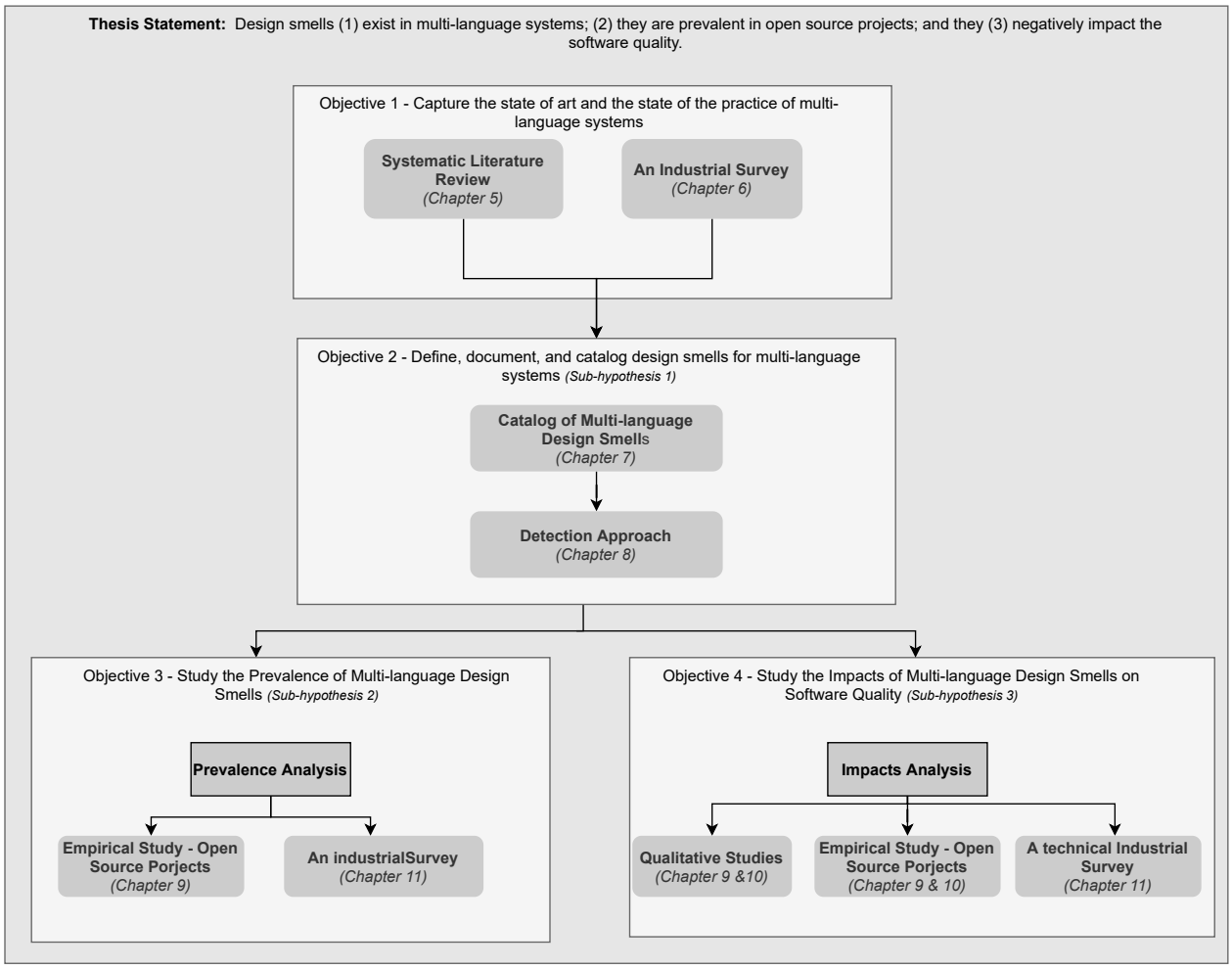


Figure 4.1 Overview of the Thesis Methodology

To address the three sub-hypothesis presented in Chapter 1, we defined four objectives as shown in Figure 4.1. The first objective is a step back allowing to get an overview of the

state-of-art and the state-of-practice *i.e.,* developers' perception of multi-language systems. Each of the three other objectives is addressing one of the three sub-hypothesis. The first objective was performed in the form of pilot studies that allowed to capture material necessary to achieve our main three objectives and thus, verify the three sub-hypothesis. We defined the four objectives as follow:

1. **Capture the state-of-art and the state-of-practice of multi-language systems.** Our goal here is to identify what is known about multi-language systems and also capture developers' perception of multi-language systems. To achieve this objective, we conducted two case studies as follows:

   (a) A Systematic Literature Review (SLR) aimed at establishing the state-of-the-art of multi-language. The goal of this study was to take a step back and capture an overview of multi-language development within the literature. Through this study we aimed to understand the motivation behind the use of multi-language development, collect the main topics and contexts of use of multi-language development, retrieve the different sets of programming languages studied in the literature, and also identify the mechanisms used to link between the programming languages. (Presented in **Chapter 5**)

   > ***First Contribution***: A Systematic Literature Review (SLR) on the state-of-the-art of multi-language programming.

   (b) An industrial survey on developers' perception of multi-language systems. The objective of this study was to capture the state-of-practice of multi-language systems *e.g.,* benefits, challenges, and any good or bad practices that developers follow or face when developing or maintaining multi-language systems. We decided to conduct a survey because we believe that surveying developers is the best method to retrieve developers' experience and practices when dealing with multi-language systems, since they are the ones who interact daily with these systems and who suffer from their challenges. (Presented in **Chapter 6**)

   > ***Second Contribution***: Technical survey from the developer's perspectives about the benefits and challenges of multi-language systems.

2. **Define, document, and catalog design smells for multi-language systems.**

   (a) A Catalogue of multi-language design smells. The two first studies reported that multi-language systems are prevalent and in continuous increase thanks to their

inherent benefits. However, such systems also introduce new and additional challenges. The two first studies also identified that one of the major concerns of multi-language systems is a lack of common guidelines, design patterns, and design smells that developers' could consider when developing multi-language systems (to mitigate the challenges of multi-language programming). Several empirical studies reported that design smells have a negative impact on the software quality and evolution [23,29,30]. However, those studies mainly target mono-language systems and do not consider the interaction between programming languages. Therefore, to support the software quality assurance of multi-language systems, we extracted, encoded, and cataloged good and bad practices in the development, maintenance, and evolution of multi-language systems. (Presented in **Chapter 7**)

> ***Third Contribution***: A catalogue of multi-language design smells.

(b) An approach to automatically detect occurrences of multi-language design smells. The detection of smells can substantially reduce the cost of maintenance and development activities. Since in our thesis, we are proposing and analyzing new types of design smells, there are no existing approach or tools to detect the occurrences of these smells. Therefore, we implemented an approach that allows the detection of the design smells defined in our proposed catalogue of multi-language design smells. The approach is applicable in the context of JNI. We focus on the analysis of JNI systems because they are commonly used by developers and also introduce several challenges [10–12]. The popularity and challenges related to JNI were also confirmed by our two first pilot studies. (Presented in **Chapter 8**)

> ***Fourth Contribution***: An approach to automatically detect multi-language design smells in the context of JNI systems.

3. **Study the prevalence of design smells described in our catalogue of multi-language design smells.** We aim through this objective to asses the prevalence of the different types of design smells defined in our catalogue of multi-language design smells from the previous objective. To achieve this objective, we performed the following empirical studies:

(a) An empirical study on the prevalence of multi-language design smells in open source projects (JNI systems). Our objective is to investigate the prevalence of the smells described in our proposed catalogue of multi-language design smells. Studies on mono-language systems, reported that occurrences of design smells may

hinder the evolution of a system by making it harder for developers to maintain the system. Thus, capturing the prevalence of our specified smells is important to understand their potential impact on the quality of multi-language systems. Hence, using our proposed approach, we detected the occurrences of multi-language design smells on a set of selected open source multi-language projects and studied their prevalence and evolution. (Presented in **Chapter 9**)

> ***Fifth Contribution***: Evaluation of the prevalence of multi-language design smells in open source projects.

(b) An industrial survey to assess the perceived relevance of the design smells by developers. We believe that developers' evaluation is important, therefore, the objective of this study is to assess how developers perceive the relevance of our specified design smells. We conducted two types of surveys, the first survey targets developers in general, while the second survey targets the original developers who contributed on files impacted by design smells from our catalogue. (Presented in **Chapter 11**)

> **Sixth Contribution**: Technical survey to capture the developers' perception and validation of our catalogue of multi-language design smells.

4. **Study the impacts of the multi-language design smells from our catalogue on software quality.** Given the known impacts of design smells on the quality of mono-language systems, it is also important to study the impacts of multi-language design smells on the quality of multi-language systems. Therefore, through this objective, we aim to capture the potential impacts of multi-language design smells on software quality. To achieve this objective, we performed the following empirical studies:

(a) Empirical Study on the impacts of multi-language design smells on software fault-proneness. Prior works reported that design smells increase the fault-proneness of mono-language systems [23,24]. Thus, since multi-language systems introduce new types of design smells, we are interested in studying the impacts of those design smells on the software fault-proneness. Therefore, in this study, we analyzed open sources projects and investigate whether files with multi-language design smells are more likely to experience bugs than files without smells. We also investigated if some specific types of multi-language design smells are more fault-prone then other types of smells. (Presented in **Chapter 9**)

(b) Empirical study on the survival of files with occurrences of multi-language smells before the occurrence of a bug. This study is complementary to the previous one. Here, we emphasize on the timeline and risk level of the introduction of bugs. Therefore, we conducted an empirical study using survival analysis to study the time to bug occurrence in files with and without multi-language design smells. We also studied the survival of specific types of smells before the introduction of bugs. Capturing the knowledge about the time to bug occurrence is useful to understand the importance of multi-language design smells and their impacts on software quality. (Presented in **Chapter 10**)

(c) A practitioner survey to assess the perceived impacts of the design smells on software quality attributes and the severity of the design smell types. In addition to the two empirical studies performed with open source projects, we also decided to conduct an empirical study using a survey because development and maintenance are manual activities performed by developers. Thus, developers' evaluations are important. Hence, the aim of this study is to capture how developers perceive the impacts of our proposed catalogue of multi-language design smells. We also aim to understand how the design smells affect different quality attributes. (Presented in **Chapter 11**)

> ***Seventh Contribution***: Empirical evaluation of the impacts of multi-language design smells on software fault-proneness.

(d) A Topic-based analysis to identify activities that are more likely to introduce bugs once performed in files with design smells. During the maintenance of a system, having knowledge of possible risky activities could help developers to reduce the risk of introducing bugs and therefore, reduce the challenges of multi-language systems. The objective of this study is to capture topics of activities that could introduce bugs. For that, we analyzed the commit messages reported as introducing bugs in multi-language smelly files. We conducted a qualitative analysis and performed topic modeling combining both manual and automated approaches to build a categorization of risky activities. (Presented in **Chapter 9**)

> ***Eighth Contribution***: Activities that are more likely to introduce bugs once performed in files with design smells.

(e) Text-based analysis to categorize bugs occurring in smelly files. To provide researchers and practitioners with deeper insights about the impacts of multi-language

smells on software bug-proneness, we examined the types and characteristics of bugs that occur frequently in multi-language smelly files. Developers often leave important information in commit logs while fixing software bugs. Such information may include an indication of the root cause of the bug and how it affects the software functionality. Therefore, our goal here is to provide a categorization and characterization of bugs related to files with occurrences of the studied multi-language smells. For that, we collected the bug-fixing commits and performed a qualitative analysis using both manual analysis and automated topic modeling to extract the categories of bugs related to multi-language smells. (Presented in **Chapter 10**)

> ***Ninth Contribution***: Categories of bugs occurring frequently in multi-language smelly files.

4.2 Chapter Summary

We presented in this chapter a detailed overview of the thesis. We described the objectives formulated to study our hypothesis. We also highlighted our main contributions, briefly described the different studies conducted during the thesis, and provided links to the respective chapters.

CHAPTER 5 PILOT STUDY 1 - A SYSTEMATIC LITERATURE REVIEW ON MULTI-LANGUAGE SYSTEMS

5.1 Chapter Overview

A Systematic Literature Review (SLR) is considered as an interesting and effective research methodology [112] to identify and discover new facts about a research area and to publish primary results to investigate specific research questions [112,113]. Since the goal of our thesis is to support the quality of multi-language systems, hence, we consider that an SLR represents the best method to collect the existing evidence on the state-of-art of multi-language systems. We decided to conduct an SLR because we believe that this type of study is important not only to provide a synthesis of available evidence about multi-language systems, but also to identify flaws and gaps in the literature, as well as important research directions.

5.2 Study Design

We present in the following our methodology to conduct the SLR on multi-language systems. We follow the procedures proposed by Kitchenham *et al.* [112] for performing systematic reviews. Figure 5.1 depicts the different steps followed to collect and analyze the data.

5.2.1 Setting the Objectives of the Study

We started by setting the objective of our study. Our objective is to investigate the state-of-art of multi-language systems within the literature and provide a comprehensive view of their usage. The perspective is that of researchers, interested in the quality of multi-language systems, and who want to get a general view on the state-of-art of multi-language systems. Also, these results can be of interest to practitioners who can benefit from knowing the different combinations of programming languages and the mechanisms used as a bridge between those programming languages. Finally, they can be of interest to managers who could have an overview about the context of use of multi-language systems and their challenges. We set out to answer the following research questions:

Part of the content of this chapter is submitted for publication as: Mouna Abidi, Manel Grichi, Foutse Khomh. "Trends of Multi-language Systems: Contrasting Academic and Practice Usage.", *Information and Software Technology (IST), 2021, Elsevier.*

RQ5.1. What is the current state of multi-language systems research publications? We aim to identify relevant research papers that discuss multi-language systems. Based on that, we extract different information that characterizes the use of multi-language systems *e.g.,* evolution over time, conference/journal publishers, etc.

RQ5.2. What are the different contexts of the use of multi-language systems in literature? We are interested to capture the main topics and contexts of use, and challenges of multi-language development in literature.

RQ5.3. What are the different sets of programming languages studied by researchers? Multi-language systems are defined by the combination of two or more programming languages. Therefore, we aim through this research question to identify the combination of programming languages frequently studied in the literature.

RQ5.4. What are the existing mechanisms used for the integration between the different programming languages in literature? As multi-language systems consist of combining programming languages with different semantics and lexical programming rules. Thus, we believe that there should be specific mechanisms that ensure the communication between programming languages. We aim to capture the mechanisms used to link between the programming languages.

5.3 SLR Design

We performed a systematic literature review covering the studies published from 2010 up to 2020 that report on multi-language systems. We performed an automated search on Engineering Village[1] to identify relevant research papers on the topic of multi-language systems.

Engineering Village is a web-based information service that provides a wide range of quality resources to collect and access researches in the context of science and engineering fields. It offers several options to refine the search queries, by applying exclusion and inclusion criteria. Engineering Village offers the flexibility to carry a research considering the period of time, language of publication, venues, specific keywords, etc. Engineering Village covers peer-reviewed engineering journals, conference proceedings, books, dissertations, workshops, etc [114]. We started by identifying relevant query keywords based on our research questions and the context of this study: **Multi-language**, **Software**, and **Analyze**. Then, for each of these keywords, we identified a set of related terms and synonyms. We also used truncation to ensure a complete collection of papers and defined the following query:

[1] `https://www.engineeringvillage.com/search/quick.url`

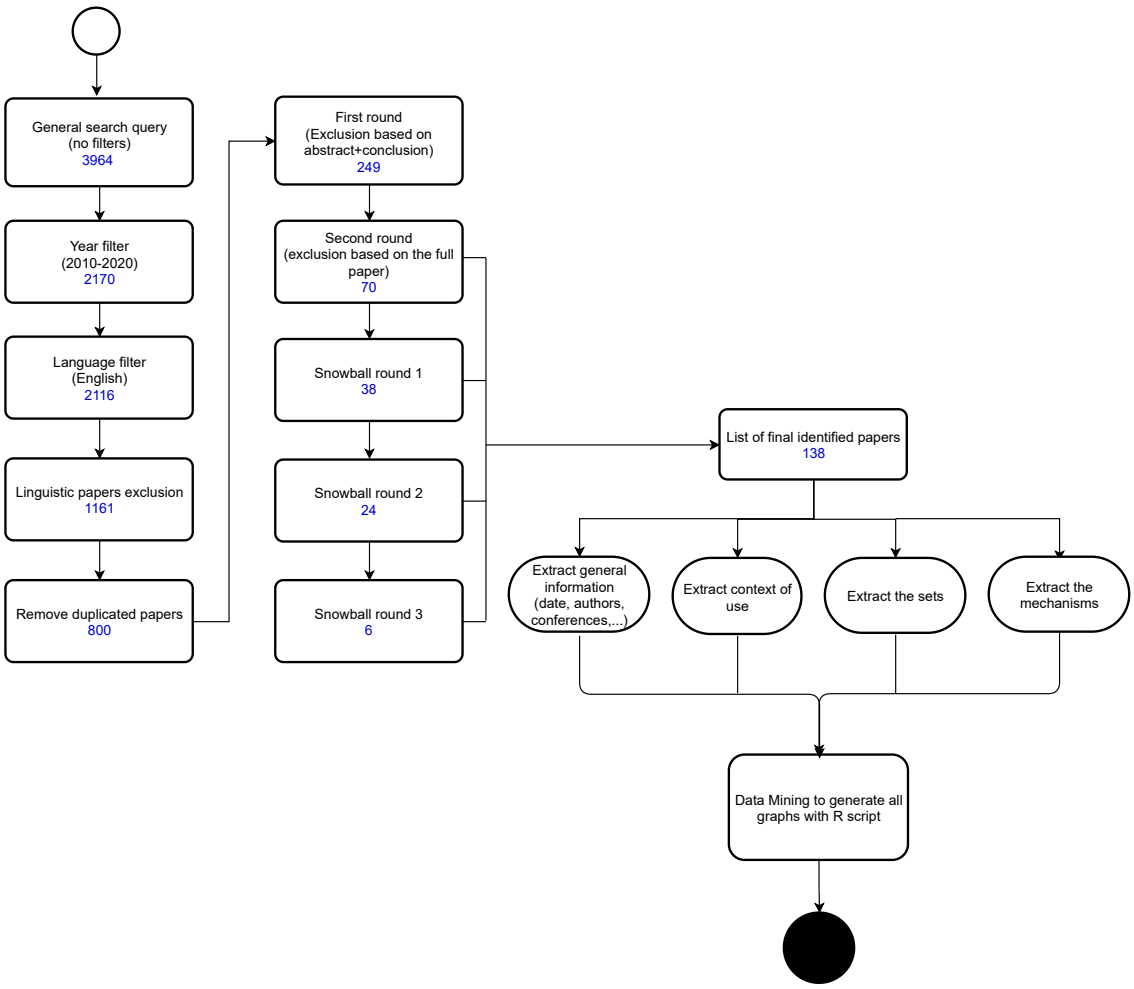


Figure 5.1 Overview of the Literature Review Process

- “Multiple language", “Multiple languages", “Multi language", “Multi languages", “Multi-language*", “Mixed language", “Mixed languages", “Mixed-language*", “Heteregenous language", “Heteregenous languages", “Polylingual", and “Polyglot".
- Software: “Software*", “Program*", and “System*".
- Analysis:“Analys*" and “Analyz*".

Then, we applied different refinement steps to get the relevant set of papers for our analysis. We limited our research scope to the title, abstract, and keywords using the option *wn KY*.

We present in the following the general query without considering any exclusion criteria. Our search queries returned a total of **3.964** references. We then performed a refining process to keep relevant papers, by considering two exclusion process, *i.e.,* an automatic exclusion and a manual analysis. Two of the research team (*i.e.,* PhD students) manually and independently

analyzed all the papers and then reconciled any differences through discussions. We present in the following the details about the refinement process.

Query 1:
((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))))

5.3.1 Automatic Analysis

The automatic refinement exclusion contains four steps.

- First, we excluded papers published before 2010. We limited the studied period to ten years (2010 until 2020) because we believe that ten years is a good and sufficient period to capture the state-of-art and get a general view about multi-language systems. The updated query returned **2170** papers out of 3964.

  Query 2:
  ((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))) AND ((2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR))))

- Second, we filtered out of the language used and excluded papers that are not written in English. The query returned **2116** papers.

  Query 3:
  ((((((((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))) AND ((2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR)) AND (english wn LA))

- Third, we observed that the keyword `Multi-language` introduces some threats and leads to confusion. It is used in the context of `Multi-language systems` but is also related to `Multi-language linguistic` field. We manually analyzed a sample of papers that contain the keyword `Multi-language` and found that it is also used in papers discussing plagiarism, speech recognition, translation, and linguistic software tools. Engineering village groups the papers according to their main field and assign codes to each group (*e.g.,* 751.5, 802.3, 932.1). Hence, based on the different options offered by Engineering village, *i.e.,* Classification code, Controlled vocabulary, and the corresponding codes, we excluded the papers considered as out of scope as presented in the query bellow. The new query resulted in **1161** papers, down from 2116.

  Query 4:
  ((((((((((("multiple languages" OR "multiple language" OR multi-language* OR "multi language" OR "multi languages" OR mixed-language* OR "mixed language" OR "mixed languages" OR "heterogeneous language" OR "heterogeneous languages" OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY) NOT (computational linguistics WN CV) NOT (cp OR ip OR ds) wn DT)) AND (english wn LA)) NOT ((751.5 OR 802.3 OR 932.1 OR 931.3 OR 801) wn CL)) AND (2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR)) NOT (linguistics OR language translation OR speech recognition OR ontology OR speech synthesis OR natural languages OR speech processing OR sentiment analysis OR teaching OR speaker recognition OR character recognition OR visual languages OR neural nets OR text detection OR behavioral research OR vocabulary OR knowledge representation OR emotion recognition) wn CV))

- Last, we removed the duplicated papers within the three data-banks (Inspec, Compendex, and Knovel) used by Engineering Village. Our final output reports a total of **800** papers *i.e.,* 621 from Compendex, 177 from Inspec, and only two from Knovel.

5.3.2 Manual Analysis

The second process of exclusions was performed by a manual analysis and verification of the 800 papers resulting from the previous step. Two members of the research team (*i.e.,* PhD students) went through the papers separately and individually to ensure a high confidence in the validity of this process. This process encompasses three steps: “Exclusion based on title, abstract, and conclusion”, “Exclusion based on the paper content”, and finally based on “Snowballing rounds”. We present in the following the details about the manual exclusion

process.

- First, we analyzed, for each of the papers in our final set of papers resulting from the previous step, the title, abstract, and conclusion to eliminate the papers that are out of scope and to keep only the relevant ones. We considered as out of scope, papers that: (1) are not directly related to multi-language systems; (2) papers that discuss multi-language systems at a very high level of abstraction (for example, papers that enumerated only examples of multi-language systems and do not provide any information that could be useful within the scope of this study); (3) papers that integrated multi-language development in databases (did not involve or combine the programming languages). Papers considered in scope are the papers that discussed multi-language development challenges, mechanisms used, multi-language tools (used, developed or extended), papers that provides a classification of the sets and combination of programming languages, etc. Through this process, we eliminated 551 and kept **249** for the next step.

- Second, we analyzed the content of the papers resulting from the previous step. Since relying only the title, abstract, and conclusion does not provide enough information to fully capture the content of the paper, we deeply studied each paper. Two of the research team (*i.e.,* PhD students) carefully read the subject papers to capture their topics and contributions in order to identify the relevant ones within the scope of this study. At the end of this step, we kept **70** papers for which we extracted the information we need to answer our research questions and achieve the objectives of this study.

- Finally, we performed a snowballing search technique to ensure high recall. We applied forward and backward snowballing techniques to minimize the risk of missing any important paper that could fit within the scope of this study. Forward snowballing technique refers to the use of the bibliographies of the subject studied papers to identify any new potential papers that could be included in this study. Backward snowballing technique refers to the identification of new papers citing the papers being considered. We iterated through the backward and forward snowballing techniques and apply for each candidate paper our inclusion and exclusion criteria described earlier. In total we did three snowballing rounds and added 70 papers. We selected from the first snowballing round **38** new papers, **24** from the second round, and **six** from the last round. We stopped the iteration snowballing process at the third round as we obtained convergence in the results and were not finding new candidates. Figure 5.1 illustrates the four last steps resulted in a final list of **138** papers. For all the subject selected papers,

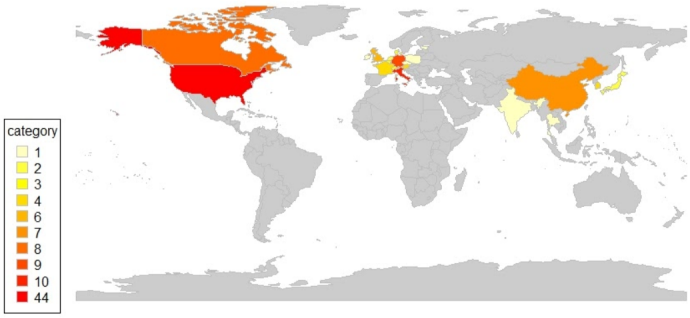


Figure 5.2 Author Affiliations (Countries)

two of the research team (*i.e.,* PhD students) carefully read the paper and extracted the following information:

- General information: author affiliation, publication years, publication countries, and conference/journal names.
- Contexts of use/challenges of multi-language systems.
- Sets and combination of programming languages.
- Mechanisms used to combine and provide the interface between the different programming languages.

5.4 Study Results

We present now the SLR report, *i.e.,* the last step in Kitchenham' methodology [112], which consists of reporting the findings.

RQ5.1. What is the current state of multi-language systems research publications?

Motivation: We aim to identify relevant research papers that discuss multi-language systems. Our goal is to extract different information that characterizes the topic of multi-language systems, *e.g.,* evolution in time, conference/journal publishers, etc.

Results: We report on 138 relevant research papers that deal with multi-language systems. From analyzing the selected papers, we report in the following a general overview about multi-language research publications:

- **The main researchers interested to the topic of multi-language systems are from USA followed by Italy, Germany, and then Canada**. Figure 5.2

Table 5.1 Number of Papers Published According to the Respective Conferences/Journals

Conferences/Journals	Occurrences
PLDI	13
ICSE	11
SCAM	9
OOPSLA	6
SANER	6
PPPJ	6
ASE	5
IWST	5
WCRE	5
SIGAda	4
Europlop	2
DLS	2
ESEM	2
EASE	2
FSE	2
POPL	2

Figure 5.3 Multi-language Systems Over Time

provides a world map overview with countries colored from dark to light following the number of papers published on the topic of multi-language systems.

- **PLDI, ICSE, and SCAM are the main conferences where papers discussing multi-language systems have been published**. We collected the main conferences and journals that published research papers on the topic of multi-language systems. Such information could help researchers to target in fact, any software engineering conference/journal that best suits their needs. We report the results of publishers in Table 5.1. We found a total of 56 conferences and journals that published one paper each, for the sake of the space, we did not include them in the Table 5.1.
- **The number of published studies on multi-language systems is increasing linearly over time**. We analyzed the evolution over time of the the selected papers and investigated how research on multi-language systems evolved over the

years. We report in Fig. 5.3, the evolution of multi-language research papers regarding the publication years. From Fig. 5.3 we can observe that the topic of multi-language systems domain presents a growing research domain, however, the results for 2020 are misleading. The process to publish papers takes a long period even after a paper acceptance. Therefore, we exclude 2020 from our conclusions, but preferred to included the papers in our results as we believe that recent research works could provide innovative and modern findings.

RQ5.2. What are the different contexts of the use of multi-language systems in literature?

Motivation: We aim to report information about the context of usage of multi-language systems, including the topics of studies available for this research area. We are also interested to capture the common challenges related to multi-language systems.

Results: The most common topic discussed in the literature in the context of multi-language systems is "Software analysis" where **'development practices" and "software dependencies" are the main challenging tasks.** To ensure a high confidence of the validity of coding, two member of the research team were involved in the coding process and then reconciled any differences through discussions. We present in the following, the steps for the coding process.

1. We started by reading all the selected papers to capture their contexts, objectives, and motivations.
2. Since our goals here are to obtain the context of use and the challenges of multi-language systems, we started by identifying the keywords/sentences that could characterize the topics studied in multi-language systems research papers (*e.g.,* code analysis, compilation, tool evolution, metric measurement, debugging, etc.). We also identified the papers that discussed challenges and issues related to multi-language systems based on keywords such as: challenge, hard, difficult, issue, ... to debug,test,develop, etc.
3. Third, based on the identified keywords resulting from the previous step, we performed a coding process to generate different topic categories. The coding process lasted two days and was performed on three steps [115]: *Open Coding*, *Axial coding*, and *Selective coding*.
4. The last step was to verify the matching between the papers and their respective assigned categories. As previously reported for the other steps, two of the research

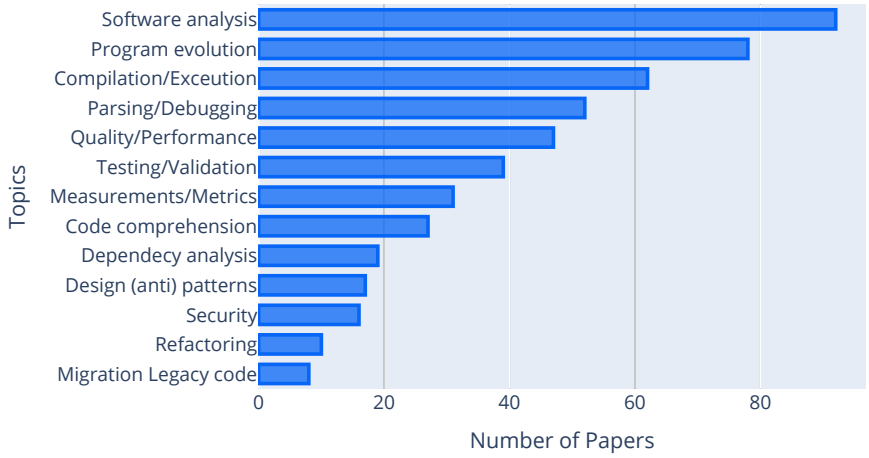


Figure 5.4 Main Topic Categories

team (*i.e.,* PhD students) were implicated and cross validated the results to reduce any possible threats related to the subjectivity of the coding process.

Figure 5.4 reports the 13 general categories of topics discussed in multi-language papers. Through our coding process, one paper can belong to more than one category if it discusses more than one topic at a time. Our results emphasize that "Software analysis of multi-language systems" is the most commonly discussed topic in the literature. Researchers are proposing approaches and tools that could help in the development of multi-language systems. This category encompasses any analysis method such as code review, slicing, static analysis, dynamic analysis, graph-based, etc.

Regarding the challenges of multi-language systems, researchers reported that multi-language systems introduce several challenges that developers face during the development and maintenance of such systems [3, 16, 17, 71]. Our results report that the most commonly discussed challenges of multi-language systems are related to software dependencies (reported in 31 papers) and lack of guidelines, *i.e.,* design patterns and design smells (reported in 28 papers). These challenges categories were reported to be related to the core of software maintenance, *i.e.,* they face daily actions such as bug fix, new requirements to add, etc. Figure 5.5 summarises the challenges of multi-language systems as reported by researchers.

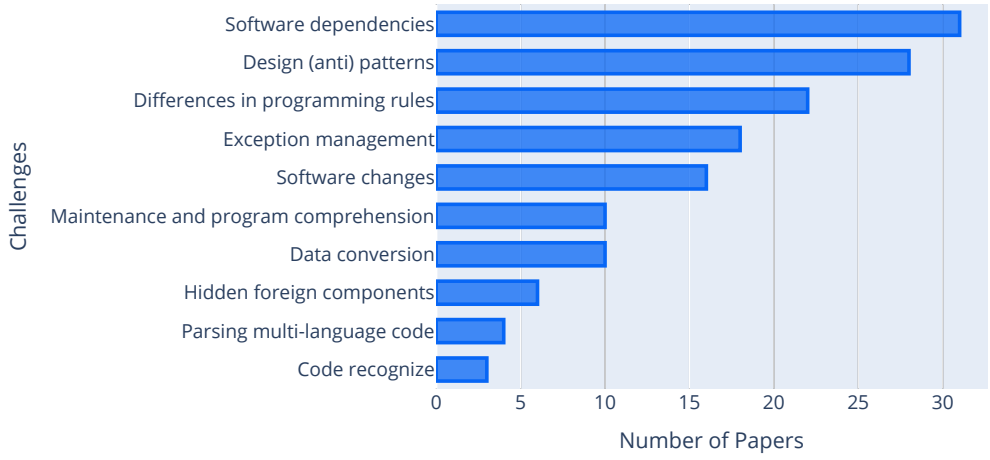


Figure 5.5 Major Challenges of Multi-language Systems

RQ5.3. What are the different sets of programming languages studied by researchers?

Motivation: Multi-language development is defined by the combination of at least two programming languages. Hence, we aim to report the existing combination of programming languages discussed in the literature. For that, we extracted from the studied papers (138 papers) all information related to the combination of programming languages.

Results: The most used set in the literature is the combination between Java and C/C++ programming languages (reported in 52 papers (37.68%)). Our results report 60 sets of programming languages (*i.e.,* combinations of programming languages). One paper can discuss more than one set of programming languages. We report in Fig. 5.6 the top 20 sets of programming languages that are the most discussed in the literature. We limited the figure to the top 20 to ensure the clarity and visibility of the figure. The combination of Java and C/C++ is the most commonly studied combination of programming languages; with 52 papers (37.68%). The combination of Java and JavaScript takes the second place with 15 papers, closely followed by the combination of Python and C (13 papers). The remaining sets of programming languages were presented in less than 5 papers (3,62%).

RQ5.4. What are the existing mechanisms used for the integration between the

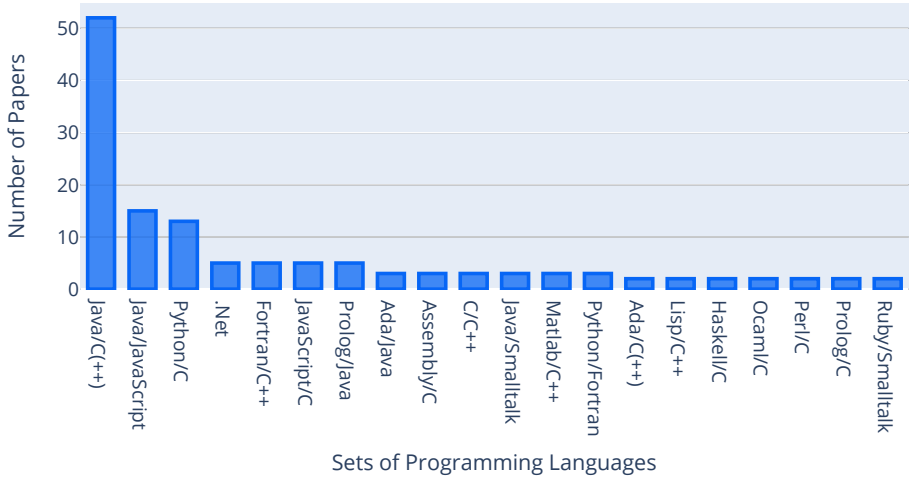


Figure 5.6 The Top 20 Combinations Of Programming Languages Discussed in Literature

different programming languages in literature?

Motivation: Since multi-language systems allows to use and call from one programming language libraries and components implemented in another programming language, there must be some specific techniques or mechanisms that ensures the communication between programming languages. Therefore, our goal is to investigate the existence of such mechanisms and their prevalence in studies discussing multi-language systems.

Results: The most discussed mechanism in the literature to integrate programming languages is Java Native Interface (JNI) with 51 occurrences (36.95%). We extracted from the studied papers the different mechanisms discussed as offering a bridge between programming languages. These mechanisms consider the interaction and combination of programming languages and could be:

- Techniques that offer a bridge between programming languages and that help developers within their multi-language development analysis such as: abstract syntax tree, graphs, etc.
- The existing interfaces used to combine between programming languages *e.g.,* JNI, Python/c, etc.
- Tools and approaches providing services for multi-language systems: dependency analysis, testing methods, debugging tasks, quality metrics, etc.

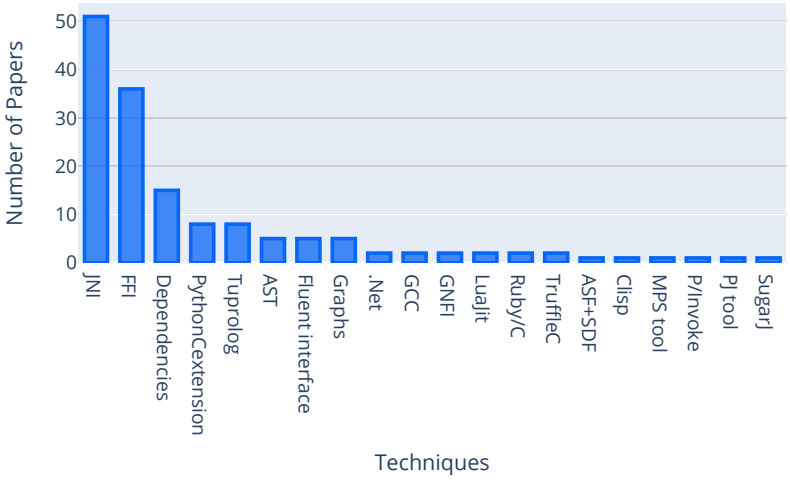


Figure 5.7 Techniques Used for the Integration of Programming Languages

- New approaches developed and proposed by researchers that target multi-language systems.

Figure 5.7 presents the most common mechanisms studied in the literature. We limited Fig.5.7 to the top 20 mechanisms for clarity and to ensure simplicity for the reader. Similarly to the previous research question, one paper can introduce more than one mechanism (method, tool, technique, etc.) at a time. Our results show that 32 papers (21.73%) did not report any details about the mechanisms used to combine and analyze programming languages. Our analysis report a total of 90 mechanisms, where the most discussed technique is Java Native Interface (JNI) with 51 occurrences (36.95%), which is aligned with the results of the previous research question, *i.e.,* the combination of Java with C/C++ is the most common studied set of programming languages. The second most discussed mechanism is Foreign Function Interface (FFI) with 36 occurrences (26.08%). In these 36 papers, researchers discussed FFI in general without specifying any combination of programming languages such as JNI or Python/C extension, or Ruby/C extension. Then, dependency analysis technique was reported in 15 papers (10.86%). Eight papers reported the use of TuProlog tool and Python/C (5.79% each). Fluent interface, graph techniques, abstract syntax tree (AST) were discussed in five papers each (3.62% each). GNU Compiler Collection (GCC), LuaJIT tool, RubyC, Microsoft .Net, GNFI, and TruffleC were reported in two papers each (1.44% each). Each of the remaining mechanisms were discussed only in one paper.

5.5 Discussion

With the increasing popularity and usage of multi-language systems, researchers are giving interest on studying such systems. Several papers discussed the benefits and also the challenges related to multi-language systems [3, 5, 16]. Researchers proposed solutions that could help to overcome those challenges and reduce the complexity of multi-language systems. Especially, researchers focused on software analysis and highlighted the need of formal guidelines to improve the quality of multi-language systems [3, 19]

From the systematic literature review, we observed that studies in the literature are mainly focusing on the combination of Java and C/C++. This could be explained by the popularity of Java, and some of its limitation that could be overcame thanks to its combination with a native code, *i.e.,* C/C++. This combination is usually performed through JNI. JNI allows to increase the performance of applications through the integration of low-level libraries for calculations and graphics. JNI also allows the use of hardware and direct operating systems commands, which opens a lot of opportunity especially for android application[2]. There is an emergence of new programming languages that are yet not well studied in the literature including python and JavaScript[3]. Therefore, we believe that researchers should also consider other sets of programming languages. Capturing the mechanisms used in the context of multi-language systems was challenging. Unfortunately, not all the papers were describing in details the methodology used to combine between programming languages or to ensure their interactions. However, we were able to extract such information by investigating the documentations we were able to access, *e.g.,* references, developers blogs, GitHub, etc.

5.6 Threats to Validity

This section discusses threats to the validity of our systematic literature review.

Threats to construct validity: The results of this study are based on our own evaluation and on the application of a defined criteria to include and exclude papers considering the scope of the study. However, to mitigate any possible threats, two PhD students were involved in this study. Those two members manually and independently analyzed all the papers, extracted the answers to each of the reported research questions, and then reconciled any differences through discussions.

[2] `https://redwerk.com/blog/3-reasons-why-we-love-jni/`

[3] `https://simpleprogrammer.com/top-10-programming-languages-learn-2018-javascript-c-python/`

Threats to internal validity: We manually validated and discussed until reaching an agreement all the keywords, the inclusion, and exclusion criteria, as well as all the findings resulting from the coding process. Through this study, we extracted the usage and main challenges related to multi-language systems discussed in the literature. We assumed that the challenges are those that developers are facing when implementing or maintaining multi-language systems. To complement the finding of this study, we conducted a survey with professional developers that we present in Chapter 6.

Threats to external validity: Regarding the generalization of our results, our study may present some threats. However, we mitigate those threats by favouring the recall over the precision at the first stage of our research query to minimise threats of missing relevant papers. However, on the next steps, we focused on the precision to keep only the papers that are within the scope of this study. We may have excluded or missed some papers that study the topic of multi-language systems between 2010 to 2020. We accept this threat, as exclusion and inclusion criteria may vary. To minimise the risk of missing important papers, we applied forward and backward snowballing process.

5.7 Chapter Summary

We reported in this chapter our results from conducting a systematic literature review on multi-language systems. We analyzed a total of 138 papers and reported that the combination of Java and C/C++ is the most commonly discussed set of programming languages in the literature. This combination is performed through JNI. We also reported that the software analysis is the most common research topic studied in multi-language research papers. Our results report code dependencies and lack of guidelines among the common challenges of multi-language systems. Therefore, from the outcome of this study, we believe that there is a need for formal guidelines, *i.e.,* design patterns and design smells for multi-language systems.

CHAPTER 6 PILOT STUDY 2 - DEVELOPERS' PERSPECTIVES ON MULTI-LANGUAGE SYSTEMS

6.1 Chapter Overview

The systematic literature review conducted in Chapter 5 pointed that despite the huge interest in multi-language systems and their popularity, these systems introduce new and additional challenges, and that there is a lack of formal guidelines to follow when dealing with such systems. Several other studies in the literature also discussed the challenges of multi-language systems, including challenges related to the complexity and the interactions among different programming languages [4]. They report that those challenges make software development, program comprehension, and maintenance activities more difficult than with mono-language systems [1–3, 16]. In this chapter, we aim to complement the findings obtained from the previous chapter and provide a practitioners' perspective on multi-language systems, their benefits, and challenges. To achieve this goal, we conduct a survey of developers. We use a survey because it is a "[s]ystem for collecting information from or about people to describe, compare, or explain their knowledge, attitudes, and behavior" [116]. Thus, by questioning the developers, we can investigate the challenges related to multi-language systems and the inherent benefits that could explain their popularity.

6.2 Study Design

We now present the methodology that we followed to perform our survey. We used Fink's guidelines [116] for setting up and conducting this survey. This methodology has also been used in a survey on mono-language code smells [29]. It includes the following steps: (1) setting objectives for information collection, (2) designing the study, (3) preparing a reliable and valid survey instrument, (4) administering the survey, and (5) managing and analyzing the data. Figure 6.1 summarises our methodology.

Part of the content of this chapter is submitted for publication as: Mouna Abidi, Manel Grichi, Foutse Khomh. "Industrial Lookout: How do Developers Deal with Multi-language Systems?", *Journal of Systems and Software (JSS), 2021, Elsevier.*

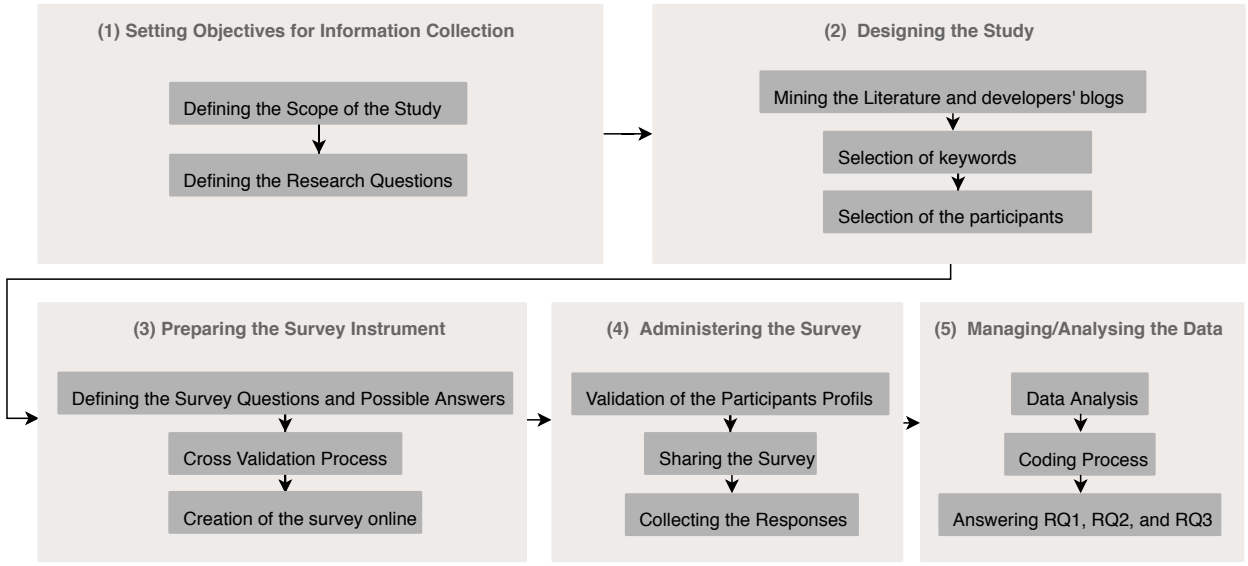


Figure 6.1 Overview of the Methodology

6.2.1 Setting Objectives for Information Collection

Our objectives are to understand and, hence, investigate how developers deal with multi-language systems, *i.e.,* identify the challenges related to multi-language systems, practices, and techniques used by developers to handle the complexity and ensure the quality of multi-language systems. The quality focus is the prevalence and usage of multi-language systems. We also examine the benefits of multi-language programming that could explain the reasons behind their increasing popularity. The context of the study consists of subjects (participants), *i.e.,* professional developers providing their perception about multi-language systems and their challenges. The perspective is that of researchers, interested in the quality of multi-language systems and who want to get evidence on benefits and challenges of such systems. Also, these results can be of interest to professional developers maintaining multi-language systems and dealing with the challenges introduced by the combination of programming languages. We defined the following research questions:

RQ6.1. What benefits do developers obtain from multi-language systems?

RQ6.2. What challenges do developers face when working with multi-language systems?

RQ6.3. What practices do developers use in multi-language systems to overcome these challenges?

6.2.2 Designing the Study

We combined open and closed questions. We used open questions to collect participants' opinions and experiences with multi-language systems as well as the practices, techniques, and tools that they use. We used closed questions to collect the participants' opinions about specific questions for which we provided predefined answers. In all questions, we allowed participants to select a "null" option: "Other (please specify)" if they do not want to answer or want to answer differently. Some questions of the survey cross-check other questions, especially for open and closed questions. The participants' information was stored separately from the survey answers to ensure their anonymity.

Participants Selection We used LinkedIn[1] as a research tool to reach potential participants because it is one of the largest, professional social networks in the world. We targeted developers with experience in multi-language systems but also with experience with more than one programming language.

6.2.3 Preparing the Survey Instrument

We used *CheckMarket*[2] to create an on-line survey. CheckMarket allows conducting fully-anonymous surveys while keeping track of participants by generating tokens in the form of Ids for each participant. We prepared the survey based on our systematic literature review of the state-of-the-art on multi-language systems presented in Chapter 5. The SLR helped us to identify the dimensions and scope of the questionnaire, the individual questions, and the possible answers for each question [61, 63, 71]. We added in the survey the references from where we extracted the possible answers to allow participants to explore these references if they are interested to have more information on a specific topic. We relied in the literature to extract the questions and their possible answers.

The survey starts with a preamble that includes the information about the research team and the ethical rights. The preamble also describes the objective of the study as well as the research questions that we aim to answer.

The survey contains in the first part three closed questions to retrieve background information about the participants and their profiles. The survey asks about their work positions, the number of years of working experiences, the domain of activities of their organisations.The second part contains open and closed questions about the use of multi-language systems in

[1] `https://www.linkedin.com/`
[2] `https://www.checkmarket.com/`

general, how it is perceived by the participants, if they believe that the use of multi-language systems is increasing or decreasing over time, the programming languages used in combination, the challenges that they face in their daily work, and how they solve these challenges. The last section also contains open and closed questions about practices in developing multi-language systems. We asked the participants if they are using specific tools/methods to deal with specific challenges incurred by nature (*e.g.,* language interactions) of multi-language systems.

Before conducting the survey, we cross-validated the questions and their answering options. The survey was designed in a way that even if the participants did not complete entirely the whole survey, we could still use their answers because each completed section answers one or some of our research questions. The total number of answers may be in some cases higher than the number of participants because they could select more than one answers. It can be lower in the case of optional questions (*i.e.,* Q5, Q7, Q10, Q11, and Q13).

6.2.4 Administering the Survey

Once we identified potential participants, we studied their profiles. We manually validated each of these profiles before contacting the participants. Our validation was based on their bibliography, summary, and previous projects. Through this validation process, we ensured to reach only professionals with at least six months of experience. We also avoided participants with no experience but that used one of these keywords as skills. We used, as inclusion criteria, keywords that can be related to multi-language systems when reading their summary and skills. As exclusion criteria, we also manually validated their LinkedIn profiles by checking their skills and previous projects. We contacted 280 developers that satisfied these criteria and reported having knowledge with different programming languages as well as practitioners' experience in software engineering in general. We invited these 280 professional developers to participate in our survey. 93 of the 280 invited participants completed the whole survey, while 40 of them did not complete the last section. Hence, we consider the total number of answers to be 133 (47.5%). The total number of responses received was 187 but we eliminated 54 surveys, in which developers completed only the first part of the survey related to their background.

6.2.5 Managing and Analyzing the Data

We extracted the responses for all the questions using absolute scales and percentages and then categorized the answers. We analyzed the responses of closed questions using numbers and percentages and then build the corresponding graphs. We analyzed the responses of open

Table 6.1 Survey Questions

N.	Survey Question
Section I Background	
Q1	What is your role within your organization?
Q2	How many years of experience do you have in software engineering?
Q3	What is the domain of activity of your organization?
Section II: Multi-language Systems	
Q4	Are you frequently using multi-language programming during your daily work?
Q5	To which domain your last multi-language project belongs to?
Q6	In your opinion, has the use of multi-language programming increased or decreased over time?
Q7	From your previous answer, why did the use of multi-language increase or decrease over time?
Q8	How do you evaluate the impact of multi-language programming on the following aspects of software development?
Q9	What are the main issues you have encountered in your multi-language project(s)?
Q10	How did you solve those issues?
Q11	Do you use any practices or patterns when developing multi-language systems?
Section III: Patterns and Practices	
Q12	Which learning sources do you use to learn multi-language patterns and practices?
Q13	Are you using any tools or methods to detect multi-language patterns and good/bad practices?
Q14	How do you verify that you have avoided the common pitfalls when using multi-language systems?
Q15	How do you evaluate the effort needed to remove bad practices of multi-language systems in those phases?

questions by coding them[3]. Similar to the coding process reported in Chapter 5, We followed the process proposed by Bluff [115] to transform the qualitative data. This process consists of assigning codes to answers so that codes can be analyzed like numerical data where each of the answers belongs to one or some predefined categories following these steps:

1. For each question, we studied all the answers provided by the participants and identified potential categories and codes. We used disjoint categories and code.
2. After identifying potential categories, we studied again the answers and started coding them, we attribute codes to each of the answers.
3. Next, we merged similar codes into the same category. If an answer could belong to more than one category, we put it in all possible categories: the total number of answers in all categories is greater than that of answers.
4. Finally, we validated the categories to ensure that the answers are in the right categories and we do not lose information when combining them.

6.3 Study Results

We now present the results of our survey. We report here the participants' answers while in Section 6.4 we use them to answer our research questions.

[3] `https://www.infosurv.com/how-to-code-open-end-survey-question-responses/`

6.3.1 Demographic Information

The respondents originate from 14 different countries which illustrate a good international coverage. 83 are from Canada, 21 from France, 11 from Tunisia, seven from the United States of America, two from the United Kingdom, two from Germany, and one from Morocco, Switzerland, Ukraine, India, Netherlands, Iran, United Arab Emirates, respectively.

We report from Q1 that participants hold different work positions in their companies: 53 (39.85 %) are software engineers, 33 (24.81%) are developers, 11 (8.27%) are team leaders, seven (5.26%) are project managers, four (3.01%) are architects, three (2.26%) are testers, three (2.26%) are self-employed, and 18 (13.53%) selected "Other" and reported: Analyst developer, CEO, DevOps Engineer, Director, Qc inspector, Researcher, Sales engineer, Systems Engineer, and Technology consultant. We observe from Q2 that participants have different years of experience: five (3.76%) have less than one year of experience, 73 (54.89 %), between one and five years 29 (21.80%) between five to ten years, 26 (19.55%) more than ten years. This diversity in the background gives us confidence that we reach a diverse group of participants. We found from Q3 that participants are working in different domains of activities. Our results report that 35 (26.32%) are in analytics (business, IT services, big data), 33 (24.81%) are in research and development, 16 are in banking and insurance (12.03%), five are in robotics and embedded systems, four are in networking (3.01%), two are in heath-care (1.50%), and two others are in games (1.50%). 36 (27.07%) of the participants selected the option "Other".

6.3.2 Developers' Perception about Multi-language Systems

Our results suggest that multi-language programming is intensively used in the Industry. Its popularity is increasing over time, thanks to its multiple benefits.

We report from Q4 that 120 (90.23%) among the 133 participants continuously use multi-language programming during their daily work. We asked in Q5 the participants about the domain of activity of their last multi-language project. This question was a single choice. Results show that 76 (56.39%) of the participants use multi-language development for client–server applications, 17 (12.78%) for mobile applications, 14 (10.53%) for desktop applications, and eight (14.29%) for embedded/operating systems. From our results we found that 18 of the participants selected the "Other" option: automation systems, business intelligence, micro services, multi-platform project, and Web applications.

We asked the participants in Q6 about their opinions and perceptions regarding the use of multi-language systems, if they believe that it increased or decreased over time. 83 (62.41%)

answered that it increased over time, 17 (12.78%) answered that it decreased, 16 (12.03%) believed that it remained the same, and 17 (12.78%) reported that it is impossible to know. The majority of the participants reported that the use of multi-language programming is more common today and that it is increasing, which confirms that today's programs are build using more than one programming language.

We asked the participants in Q7 to explain their previous answers. After the coding process discussed in Section 6.2, we report that participants justified the increase because: complex systems need more than one language (30.3%), advantages of different languages (28.78%), emergence of new languages/convergence of technologies (16.66%), reuse of code (15.15%), flexibility/agility/interoperability (4.54%), and migration of projects (4.54%). Participants who thought that the use of multi-language systems were decreasing justified their perception by lacks of understandability (31.25%), maintainability (25%), compatibility (12.5%), by the difficulty of integration (12.5%), increase of complexity (6.25%), interoperability (6.25%) and, waste of time to train developers (6.25%). We can observe from the results that the use of multi-language programming presents different appealing advantages for developers; mainly they facilitate the implementation of complex systems where each language can be used for specific purpose. Other reasons include taking the benefit from each language and reuse of code. These systems present also some drawbacks as their usage introduces some challenges including the understandability, maintainability, and increase of complexity.

6.3.3 Developers' Perception about the Impact of Combining Programming Languages

Developers reported that multi-language systems present several benefits. However, they also reported facing many challenges when dealing with multi-language systems.

We asked the participants in Q8 to evaluate the impact of using more than one programming language on seven aspects of software development. We found that the use of more than one programming language is perceived to positively impact system performance and architecture, the implementation of the initial code, and the developers' motivation but it is perceived to negatively impact the software understandability. Figure 6.2 presents the obtained results regarding the impact of the use of multiple programming languages. Participants report a neutral impact on memory usage and the translation of requirements to code. We could explain the positive impact on system performance, architecture, and implementation of initial code because, as reported in Q7, developers often leverage the strengths of multiple languages to cope with challenges of building complex software. Regarding de-

velopers' motivation, we believe that it is related to their involvement in different projects and the requirement to learn new programming languages and technologies. The reuse of existing code allows developers to avoid re-implementing the code, which can have a positive impact on their motivation.

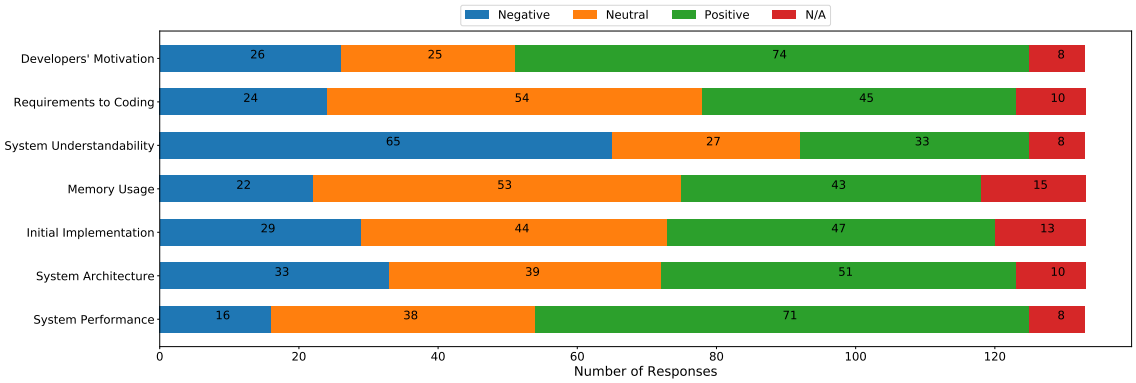


Figure 6.2 Impact of the Use of Multi-language Systems

We asked the participants in Q9 about the main challenges of multi-language programming. Results show that 85 (63.91%) participants faced maintenance problems, 40 (30.31%) performance problems, 34 (25.56%) robustness problems, 28 (21.05%) security problems, and 24 (18.05%) correctness problems. Besides the suggested challenges, participants could add additional challenges. Other answers reported challenges related to software understandability, knowledge about the different languages, compatibility between languages, threading, and interoperability.

We surveyed the participants in Q10 about how they solved the challenges reported above. Only 60 (45.11%) participants answered this question. Results show that 14 (23.33%) of the participants rely on teamwork to solve these challenges and refer to the right person for the right language, 13 (21.67%) use tests and debugging, nine (15%) use design patterns and avoid design smells specific to each programming language, eight (13.33%) are still dealing with issues and avoid multi-language programming when possible. Results show that dealing with multi-language systems challenges often require involving developers with different expertise in the required programming languages. Developers also refer to mono-language design patterns when they consider each language separately and *not* the interactions among languages. Some participants also reported that specific tools can solve some of these challenges. Six (10%) of the participants reported applying re-engineering techniques and refactoring to solve the challenges, four (6.67%) expressed that developers' experience has a direct impact to solve the issues, four (6.67%) reported the use of code review, three (5%) expressed that they always rely on documentation, three (5%) explained that they minimise the interactions

between different languages. Surprisingly, only two (3.33%) reported using specific tools to deal with the issues and assess the quality of their multi-language systems. Accessing tools and guidelines could help developers handle the challenges.

To better understand how multi-language systems is perceived by developers. We analyzed the answers to the open questions using coding techniques, following by a grounded theory [117]. These categories can open new perspectives of research to better investigate multi-language systems. We defined five categories as follows:

- **Challenging:** This category encloses cases where participants expressed having problems understanding multi-language systems and where multi-language systems introduces several challenges. Examples included in this category: maintainability, understandability, complexity, compatibility, integration, etc. (*e.g.,* `"Maintenance is difficult, integration is complex", "Usage of multiple languages inevitably adds complexity hence a more difficult maintenance"`).
- **Beneficial:** This category includes statements reported by the participants that expressed several benefits related to multi-language systems; including: benefits from each language, reuse of code, flexibility, and interoperability (*e.g.,* `"more advantages and it is very helpful for a developer. Using multiple language we gain a lot of time and we reuse what exist instead of reinventing the wheel"`).
- **Team work:** This category encompasses cases where participants explained that they work in team to better succeed and cope with multi-language systems challenges. It also includes cases where developers expressed relying on the right person for the right language (*e.g.,* `"Having different experts in different languages inside a single team"`).
- **Lack of multi-language systems dedicated support:** This category is composed of statements related to lack of tools and guidelines to deal with multi-language systems. Some participants expressed that they are avoiding as much as possible the use of multi-language systems (*e.g.,* `"The problems are only mitigated and not solved"`).
- **Avoidance when possible:** This category groups cases where participants explicitly expressed that, when possible, they avoid the use of multi-language systems. Most of them explain their choice by the complexity and bugs related to the combination of programming languages (*e.g.,* `"We attempt to discourage the introduction of new languages". "Not solved... limited use of multiple languages"`).

6.3.4 Multi-language Practices

The participants reported that they rely on practices defined for mono-language systems.

We surveyed the participants in Q11 and asked if they are following any specific patterns or practices during the development of multi-language systems. 32 (24.06%) of the participants answered this question. After the coding steps, the results show that 26 (19.55%) are using design/architectural patterns and standard good practices for each language. They do not use any pattern and–or practice specifically designed for multi-language systems. Some of them expressed that their current practices are not enough to handle the complexity of multi-language systems and called for specific practices for multi-language systems. While three of the participants rely on the project architectural specification. Only two (1.5%) of the participants reported using code review in their multi-language development and another two participants using tests at each stage to ensure the quality of their multi-language systems. Some practices were not reported due to private information (*e.g.,* `"Not anything I can disclose at this time"`).

We asked participants in Q12 about the learning sources for multi-language systems' good and bad practices. Results show that 70 (75.27%) participants relied on Stack Overflow, 54 (58.06%) in blogs, 51 (54.84%) discussion forms, 42 (45.16%) reported using Youtube and Video tutorials, and only 34 (36.56%) reported using books and articles. 11 (11.83%) selected the option "Other" and reported other sources: learning by doing, work, GitHub, online courses.

We asked the participants in Q13 if they are using tools or approaches to automatically detect occurrences of multi-language systems good/bad practices. We coded the answers following the same steps as explained in Section 6.2. Results show that only a few of the participants are using automated tools. Three of them are using mono-language patterns detection tools, two of them are using linting tools, and two reported running tests.

We asked participants in Q14 how they verify that they avoid common bad practices and how they assess the quality of their multi-language systems. Results show that only two (2.15%) of them use dedicated tools; 11 (11.83%) reported using different kind of tests, including unit and integration tests. Table 6.2 summarises the techniques used by developers to ensure the quality of their multi-language systems.

We also surveyed the participants in Q15 about the effort spent to solve the challenges and improve the quality of multi-language systems. Our survey results show that it is always a challenging task that requires medium or high effort on all phases of the software development

Table 6.2 Techniques used to Ensure Quality of Multi-language Systems

Learning Sources	Percentage
Performing code reviews	79.57%
Validation of the code against specification	60.22%
analyzing the method trace	32.26%
Generating dumps	15.05%
Using dedicated tools	2.15%
Other, please specify	11.83%

process, especially in the testing and maintenance phases. Figure 6.3 summarises the effort estimated to solve the challenges and improve the quality of multi-language systems on six phases of the software development life cycle. We believe that it is important to formulate common guidelines and practices that can be used in the development of multi-language systems.

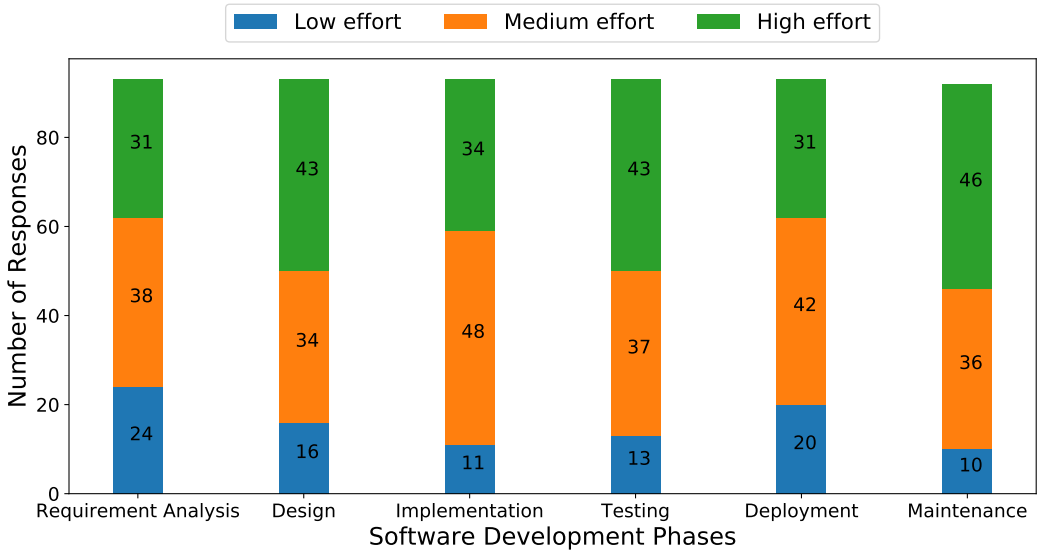


Figure 6.3 Effort Estimated to Remove Bad Practices

6.4 Discussion

We now discuss the study results shown in Section 6.3, answering our research questions. We also formulate some recommendations for multi-language systems developers.

RQ6.1: What benefits do developers obtain from multi-language systems?

First, we investigated how multi-language systems are perceived by developers. Most of the participants reported that the use of multi-language systems is increasing over the years, which validates the findings of previous studies [3, 5, 71] reporting that the use of multi-

language development is prevalent. This finding is not surprising because of the expected benefits of using the right programming language for the right problem, as expressed by one of the participants: `"Every language has its own uses"`. Our study presents the results of surveying a subset of developers that mainly reported to continuously use multi-language systems during their daily development work (90.23%). Thus, our results support that multi-language systems developers perceive that the multi-language systems usage is increasing over time. Results show that multi-programming is mainly used in client/server and mobile applications, which is not a surprising result as well, because these applications generally involve many languages and technologies. `"It is imperative to make it easier for users to access software: either via browser or mobile app while the server must carry the heavy lifting. The two sides are so different than one programming language cannot include libraries for all purposes."` We asked participants in Q7 about what they perceive to be the benefits of multi-language systems and their responses confirm what has been discussed in the literature: complex systems need more than one programming language and developers combine languages to benefit from the advantage of each one. As expressed by some participants: `"Systems are getting more complex and thus involve more technologies"` and `"There are more and more languages with their pros and cons so if you prefer the opportunities given by one specific language in a specific domain you can combine it with another language"`. Ten (15.15%) participants attributed the increase of multi-language development to code reuse (*e.g.,* `"there are more and more micro service which can be coded in any language and reused from any other languages"`.The reuse of code in multi-language systems helps developers avoid reinventing the wheel, avoid re-writing code from scratch in another programming language, which contributes in saving development time (*e.g.,* `"Using multiple languages we gain a lot of time and we reuse what exist instead of reinventing the wheel"`).

Secondly, we asked in Q8 about the impact on software quality. Multi-language programming is reported to have a positive impact on the performance of the systems. Systems using JNI, developed with Java and C++, are considered to benefit from the high performance of Java for the execution speed and the management of exceptions that do not exist in C, for example. Participants also expressed through their answers that the use of multi-language programming has a positive impact on their motivation. We can explain this result by the fact that first, including different APIs, libraries, etc., in software, written in another programming language, allows developers to reuse code, hence reducing development effort. (*e.g.,* `"the multiple language has more advantages and it is very helpful for a developer"`, `"Using multiple language we gain a lot of time and we reuse what exist instead of reinventing the wheel"`). This can impact their motivation

positively. Second, each language brings benefits as it is more adequate for certain specific tasks. (*e.g.,* `"As systems grow in size and complexity, one language cannot resolve all the possible problem out there. Hence, using a language that fits best the problem at hand is common these days"`). Thus developers can choose the right language for their tasks instead of dealing with a single language to implement all the requirements (*e.g.,* `"More languages are available, developers bring along preferences with them when contributing to components"`).

> ***Summary of findings (RQ6.1)***: The use of multi-language systems is in continuous increase thanks to several advantages, including their positive impact on developers' motivation, implementation of the initial code, reuse of code, and combined advantages/benefits provided by the use of several languages.

RQ6.2: What challenges do developers face when working with multi-language systems?

Regarding the challenges and issues of multi-language systems, from Q7, Q8, and Q9, most of the answers were addressing understandability and maintainability (*e.g.,* `"You need some extra time to understand the code", "Sometime it accelerates the development. Sometimes it complicates things and make maintainability very hard"`). These results confirm previous insights reported by multi-language programming researchers [2, 3, 16, 71]. Some of the participants pointed out that understandability depends on developers' training. They mentioned that the use of multi-language systems requires initial training for developers which takes time, but leads to better results when coping with such systems. Even if the usage of multi-language systems is increasing, some developers reported that they avoid as much as possible the use of multi-language systems due to unsolved issues such as the incompatibility and interfaces between the languages (*e.g.,* `"Usage of multiple languages inevitably adds complexity hence a more difficult maintenance", "Not solved... limited use of multiple languages", "I dealt with the consequences of these issues, but I have not solved the issues themselves"`). Multi-language systems introduce complexity due to the combination of components written in different programming languages (*e.g.,* `"Maintenance is difficult, integration is complex"`). It was also pointed out that good tool support can solve (some of) these problems. Although combining multiple programming languages in the same software can be considered a wise practice, but each good practice, if it is not well used, can introduce some drawbacks. The same software with components written in multiple programming languages can be hard to understand

since it requires competences in diverse programming languages (*e.g.,* `"Usage of multiple languages inevitably adds complexity resulting in more difficult maintenance"`). A developer who is expert in only a specific programming language can have difficulty understanding the dependencies between components written in multiple languages (*e.g.,* `"It requires a lot of effort for newcomers to understand the system"`). During maintenance activities, each language often has to be analyzed and tested differently, which can also be challenging for developers with limited knowledge on some programming languages.

> ***Summary of findings (RQ6.2)***: The challenges that developers face in multi-language systems development are mostly related to the complexity resulting from the nature of multi-language systems. Developers must have competences in diverse languages, understand dependencies between components written in different languages.

RQ6.3: What practices do developers use in multi-language systems to overcome these challenges?

One of our main results from Q10 and Q11 shows that developers rely on mono-language design patterns and standard practices to develop and maintain multi-language system (*e.g.,* `"Same as if not multi-languages projects", "using the same object-oriented practices"`). These patterns and practices are only related to each language separately without considering the coupling and interaction between languages. Even for the tools used to assess the quality of such systems, a few number of participants reported using tools. Most of these tools do not consider multi-language dependencies, or are limited to a single language (*e.g.,* `"JDeodorant to detect anomalies and refactor the code", "Ptidej to detect design patterns"`). **We believe that adequate patterns, as well as tools, must be used to solve the challenges of multi-language systems (*e.g.,* `"practices and tools for multiple language may help developers keep their code clean and maintainable"`)**. Multi-language systems even if they contain diverse components written in different languages must be analyzed as the same entity and–or the same system. If we take the case of JNI systems, it is not appropriate to analyze Java and C parts of the system separately. In JNI, the native signature will be declared in the Java side and the implementation in the C side. Therefore, the analysis must be performed on the whole system. Developers should take into consideration the link between the different programming languages when dealing with multi-language systems. Most of the participants are using *StackOverflow*, *Blogs*, and *Discussion Forms* to learn about multi-language systems Patterns and Practices. This

can be explained because these practices are not formally documented. It is important to have formal and popular websites defining the multi-language systems patterns and practices. As for the case of mono-language systems, documentation is easily accessible and not scattered between developers' documentation and StackOverflow. We can easily search for any of the mono-language design patterns or anti-patterns in books [27,79] or access websites such as *tutorialspoint*[4], *sourcemaking*[5], *sourcemaking*[6], etc. Tools are also available for mono-languages design patterns and anti-patterns including *Ptidej, JDeodorant, DECOR*, etc. We believe that formal guidelines and tools could help developers to cope with multi-language systems challenges as expressed by our participants (*e.g.,* `"good practices and tools for multiple language may help developers keep their code clean and maintainable"`).

Summary of findings (RQ6.3): Developers are mainly relying on mono-language patterns/practices and teamwork to solve the challenges of multi-language systems. They also usually use different testing methods to handle the complexity of multi-language systems either in developing or understanding their source code.

6.5 Threats To Validity

In this section, we discuss the threats to the validity of our study [118].

Threats to Construct Validity According to Fink, a survey is used to collect information from or about people to describe, compare, or explain their knowledge on a specific topic [116]. Thus, by questioning developers, we can investigate the challenges related to multi-language systems and capture the developers' practices as well. We followed formal guidelines [116] to conduct our survey as described in Section 6.2. We relied on materials from the literature to design the survey and analyze the results [11, 29, 63].

Threats to Internal Validity Acquiescence bias is a kind of response bias where respondents agree with all the questions in the survey or misunderstand the questions. To minimise these threats, we adapted our survey to the target population, *i.e.,* developers. We provided, in the survey preamble, the definition of all the terms used.We allowed participants to select the options "N/A" and "Other (please specify)".

[4] `https://www.tutorialspoint.com/design_pattern/index.htm`
[5] `https://sourcemaking.com/design_patterns`
[6] `https://refactoring.guru/design-patterns`

Threats to External Validity Participants might not be representative of the general population of multi-language developers. Thus, the generalisability of our survey might be limited. To mitigate this threat, we advertised our survey through LinkedIn, which is one of the largest professional networking site[7]. We targeted developers from different backgrounds.

Threats to Conclusion Validity The choice of material used to conduct this study and analyze the data is not impacting the results. Consequently, any other tools will likely provide the same results. We diversified the keywords and tried to reach participants with different backgrounds and skills. We cross-validated our results analysis and coding methodology.

Threats to Reliability Validity We mitigate the threat to reliability by providing online access to all the data used to conduct this study. We provided in Section 6.2 the information needed to replicate this study. However, the list of potential participants could vary. Thus, it would be difficult to reproduce the same list of profiles even using the same keywords.

6.6 Chapter Summary

From surveying the developers, we found that multi-language programming is used extensively in the industry. The number of multi-language systems seems to increase over time as it presents different benefits. However, multi-language programming seems to increase the complexity of software systems. Developers reported that they face many challenges when dealing with multi-language systems. From our survey analysis, we found that even if these systems introduce several challenges as discussed in the literature, they also present benefits that explain their popularity. For that we recommend to deeply study their quality and help the developers dealing with such systems. Our results also point that practices and tools to support multi-language systems, must be developed to assess and support the quality of these systems. The findings from this chapter motivated us to extract and document formal guidelines for multi-language systems as presented in Chapter 7.

[7] `https://www.forbes.com/sites/jaysondemers/2015/07/22/10-reasons-your-brand-needs-to-be-on-linkedin/#2a2ea9b63aca`

CHAPTER 7 A CATALOGUE OF MULTI-LANGUAGE DESIGN SMELLS

7.1 Chapter Overview

Our findings from the two previous chapters suggest that, despite the huge interest in multi-language systems and their popularity, these systems introduce new and additional challenges and that there is a lack of formal guidelines that developers could consider when dealing with multi-language systems. Therefore, to support the software quality assurance of multi-language systems, we extracted, encoded, and cataloged good and bad practices in the development, maintenance, and evolution of multi-language systems. We analyzed the source code of open-source multi-language systems as well as developers' documentation, and programming-language specifications. These systems contain mainly Java/C/C++ but also include other programming languages *e.g.,* Python, JavaScript, Lua, etc. We identified good and bad practices in the code as well as issues reported in the developers' documentation and bug reports. We encoded and cataloged these observed practices and report our findings in the form of design smells. These design smells could apply to microservices or, rather, to the implementation of microservices, to database and procedural programming languages as to any other pieces of code in which such poor design or implementation choice could appear. We have focused our efforts on design smells to complement the previous work on identifying design patterns for multi-language systems [14, 15, 83, 119]. Both design patterns and design smells are important to improve the quality of multi-languages systems and help developers coping with their challenges. In this chapter, we present the proposed catalogue of multi-language design smells studied in this thesis.

7.2 Study Design

We present the steps followed to collect and document the proposed multi-language design smells. Figure 7.1 presents an overview of our methodology. We believe that the following steps could be used for a replication purpose as well as for any future study investigating new design patterns and design smells.

Part of the content of this chapter is published in: Mouna Abidi, Foutse Khomh, Yann-Gaël Guéhéneuc. "Anti-patterns for Multi-language Systems", and Mouna Abidi, Manel Grichi, Foutse Khomh, Yann-Gaël Guéhéneuc. " design smells for Multi-language Systems", *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP19).*

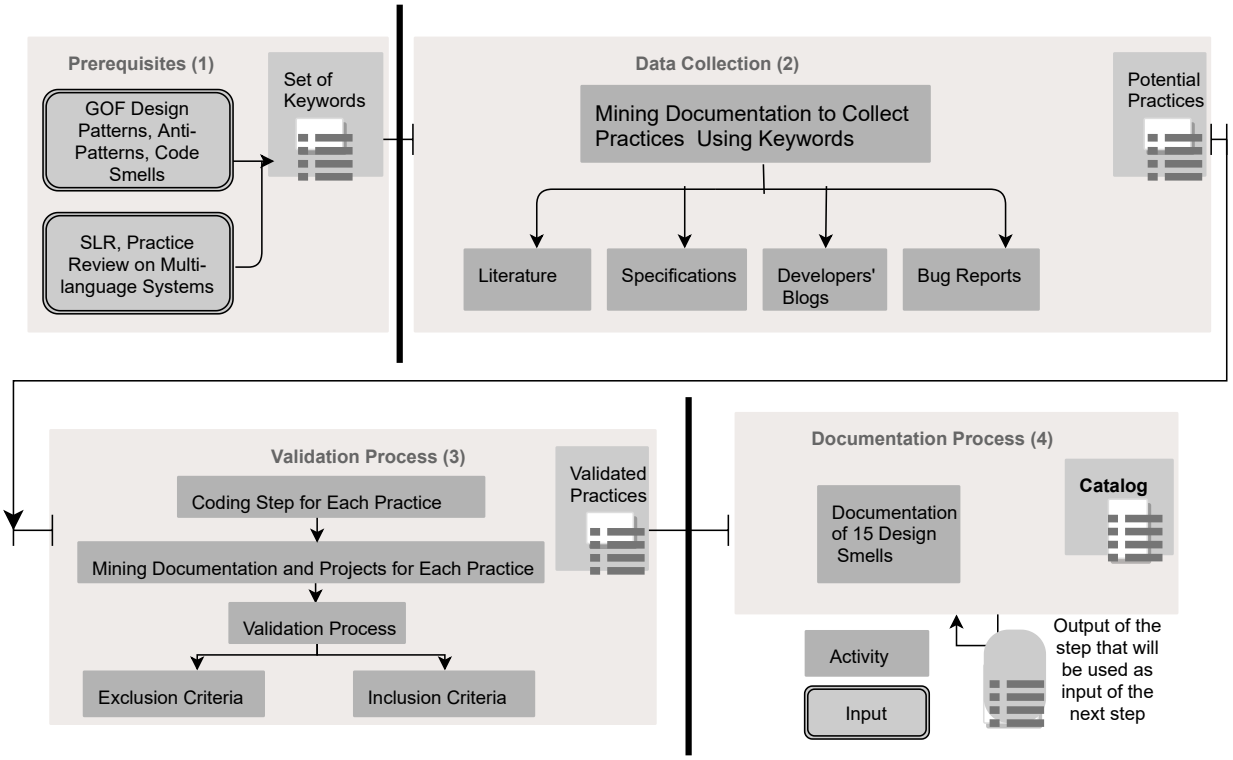


Figure 7.1 Overview of the Methodology Used to Collect and Document Design Smells for Multi-language systems.

Prerequisites: We believe that to investigate and collect design smells for multi-language systems, it is important to have enough knowledge and experience with both multi-language systems and design smells. From previous studies and our literature review presented in Chapter 3, we already had good background and knowledge on design design smells for mono-language systems. We performed a systematic literature review to investigate the usage of multi-language systems in the literature as presented in Chapter 5. These steps gave us good knowledge about multi-language systems and their challenges as well as design smells. It helped us to collect some keywords that can be used to retrieve challenges and issues related to multi-language systems. It also allowed us to better distinguish between a simple habit and a possible pattern.

Data Collection: Once we decided to collect and document design smells for multi-language systems. We started by mining all possible sources of documentation. We searched in the literature, language specification, developers' blog as well as bug reports. From our systematic literature review on multi-language systems presented in Chapter 5, we found that the most studied combination of languages is Java/C/C++. For this reason, we decided

to start with this combination. We deeply read the Java Native Interface specification [41] as well as developers' documentation to collect common practices and guidelines related to the JNI and multi-language systems. We searched in Google as well for JNI practices and find a couple of developers' blog and documentation that discuss common good and bad practices[2] [1]. We documented the practices extracted in terms of definition, context, and examples.

We then considered multi-language systems in general and searched for any other possible issues related to a combination of more than one programming languages. We analyzed bug reports and developers' documentation to extract the issues related to multi-language systems that have been reported by developers. We relied on often used websites such us *Stack Overflow*, *GitHub issues*, *Bugzilla*, *IBM Developers*[1], and *developer.android*[2]. We queried the developers' documentation and bug reports by searching for common keywords that reflect issues in multi-language systems. We relied on some keywords collected from our literature review as well as common issues reported by developers. We used the set of keywords extracted from the previous step including *JNI issue, Python/C issue, foreign library, API, polyglot, programming languages issues, incompatibility, compilation errors, memory issues, performance issues, security issues, foreign function interface.* As an example, when searching for *JNI issue* in *Bugzilla*, our query returned 23 results, among them we considered only two as possible bad practices. One was related to the library loading, the other was related to the management of exceptions[3] [4]. From *Stack Overflow*, for the keywords *JNI issue* and *Python/C issue*, we had for each of these keywords 500 results, we searched manually only for issues that have been already discussed in the developers' documentation[2] [1].

We documented all of the reported issues and possible practices in our list of potential practices. This list is then used as input to the next step, which is the validation process. We believe that our research method to collect practices was not exhaustive and that there are many other design smells that can be extracted as well. As future work, we plan to extract more practices from these sources. We considered as practices a common situation that was reported more than three times in any kind of documentation, including literature, the developers' documentation or bug report.

Data Validation: For each of the practices reported in our list of potential practices, we performed a coding process in which, we provided a definition and explanation of the practice. The explanation was in term of what are the contexts, situations, and possible

[1] `https://www.ibm.com/developerworks/library/j-jni/index.html`
[2] `https://developer.android.com/training/articles/perf-jni`
[3] `https://bugzilla.redhat.com/show_bug.cgi?id=529919`
[4] `https://bugzilla.redhat.com/show_bug.cgi?id=1045623`

examples that we should look for to retrieve occurrences of the practices. We performed a discussion between the research team to validate the explanations provided for the potential practices. We performed the validation process through different sources of information following inclusion and exclusion criteria. Through this step, we aimed to verify if the potential reported practices have been used or discussed in at least three situations and-or examples in open source systems. We searched for occurrences of these practices in different sources of information (*e.g.,* GitHub, Developers' Blog, Bug Report).

We defined a set of inclusion and exclusion criteria. As inclusion criteria, we considered a practice that was discussed in at least three situations. In the case of literature or any other type of documentation, we searched for similar situations that have been reported by developers, discussed in bug reports or developers' blog. Another inclusion criteria was applied when analyzing the source code of multi-language systems. We searched if the good or bad practices discussed in the literature were present in at least three classes, source code files, or systems. As exclusion criteria, we considered a practice for which, we were not able to find at least three of its occurrences in any of the sources of information, including open source systems. We also considered as exclusion criteria practices that seem more likely to be a simple habit than a potential design smell.

We manually searched in the source of information (*e.g.,* bug reports, developers' blog, developers' documentation) to find at least three situations where the potential practices were discussed. We also used data already extracted from one of our prior studies focusing on JNI usage [120]. The data consists of 100 multi-language open source systems. We extracted these systems from OpenHub using Python Scripts. We then downloaded the projects and manually analyzed their source code. These systems contained not only Java/C/C++ but also Python, JavaScript, Lua, etc. (*e.g., OpenCv* is mainly written in C/C++ but contains 25.239 Python lines of code, 24.427 Java lines of code, and other languages). Here are some systems on which we mainly focused during this study: *libgdx, Google toolkit, Openj9, Rocksdb, JMonkeyEng, OpenVRML, PortAudio Java Bindings, jpostal, JavaSMT, Jna, ZMQ, reactNative, Telegram, OpenCV, Tenserflow, JatoVM, SQLlite, Frostwire, Godot, python-telegram-bot.*

We manually and qualitatively analyzed the source code of the multi-language systems collected from GitHub to extract occurrences of good and/or bad practices. We also checked if the common guidelines and practices reported in the literature are followed by the developers in practice. Most of them were not, in that case, we reported it as a possible bad practice. In our case, we considered as multi-language systems practices a piece of code that is participating in the interaction between two or more languages and that has been documented in the literature as bad practice or that have been reported in bug reports or developers'

documentation as causing issues or negatively impacting the system. We considered as practices a similar situation that were observed more than three times and was discussed in the programming language specification, or developers documentation as being a bad practice.

Design Smells Documentation: We reported all the observed practices and performed a discussion between the research team to validate if the practices are really valuable and should be documented in form of design smells, or if they are only a simple practice or developers' habit. We discussed each case until a consensus was reached. We used an available template to documented our results in the form of design smells. We believe that these design smells could help researchers and developers to cope with the challenges introduced by multi-language systems.

Encoding and Cataloguing: There exist several templates in the literature to encode design smells [27, 28]. The template that we used in this thesis is inspired by Brown's template [27]. We adapted the template to the specificity of our work as follows:

- **Design Smell:** We describe in the title the name to identify the design smell.
- **Overview:** Provides a general overview about the design smell.
- **Context:** The context in which this particular design smell applies, for example, real-time systems or communication systems.
- **Problem:** It introduces the initial problem that is being solved or the problem that may lead to the wrong solution. It can be illustrated by a simple concrete example.
- **Bad Solution:** The bad Solution is the solution solving the problem on first thought but that has other negative impacts on software quality.
- **Consequences of the Design Smell:** These consequences describe the impact of applying the "poor" solution to solve the problem.
- **Refactoring:** This solution(s) presents a better solution that can be applied to remove the design smells. It includes the steps that can be followed to apply the solution.
- **Benefits of the Refactoring:** These benefits are the consequences of applying the refactoring to remove the occurrences of the design smells.
- **Related Design Smells:** If any, specify the names of related design smells.

- **Related Patterns:** If any, specify the names of patterns that could be applied in the refactored solution. In future work, we will examine the effectiveness of those solutions.
- **Examples:** These examples provide code and–or diagrams showing the design smell in context. We also provide a detailed description to illustrate the situation.

Design Smells Validation: The catalog went through rounds of a shepherding process in a pattern conference (*i.e.,* EuroPlop). In this process, an expert on patterns provided three rounds of meaningful comments to refine and improve the patterns. The catalog then went through the writers' workshop process, in which five researchers from the pattern community had two weeks before the writers' session to carefully read the catalog and provide detailed comments for each defined smell. The catalog was then discussed during three sessions of two hours each. During these sessions, each smell was examined in detail along with their definition and concrete examples. The conference chair also provided additional comments to further improve the catalogue.

7.3 Design Smells for Multi-language Systems

In this section, we present our catalogue of design smells following the template detailed previously.

Excessive Inter-language Communication

- **Overview:** A wrong partitioning in components written in different programming languages leads to many calls in one way or the other. This may add complexity, increase the execution time, and may indicate a bad separation of concerns. Occurrences of this design smell could be observed in systems involving different layers or components. For example, the same object could be used and-or modified by multiple components. An excessive call of native code within the same class, could be illustrated whether by having too many native method calls in the same class or having the native method call within a large range loop.
- **Context:** Supposing we are in a context in which we must implement a task or add a new feature that is already available as a library implemented in another language. This can also be illustrated with a situation when a single language is not suitable to implement all the tasks. It may appear in the context of embedded systems, or systems in which we need an important number of communication between different layers or modules of the application.

- **Problem: Some projects may require an important communication between components written in different programming languages. Other projects may be integrated with other modules with high reuse of features which results in several calls.** The problem is that developers or maintainers do not always know how to deal with such communication between heterogeneous components, which may lead to excessive inter-language communication. Usually, different teams may be involved separately to contribute to these components in a way that developers do not have enough knowledge about the whole architecture of the system.

- **Bad Solution:** The bad solution would be, to add the foreign code and access features from one language to another each time in the program we need to access foreign objects without considering the number of calls from one language to another.

- **Consequences of the Design Smell:** This design smell will result in an excessive passage of objects and calls between the host and the foreign language. In a study focusing on JNI systems, they found that calling the native code from Java code can take five times longer than a regular method call[1]. Similarly, calling Java code from the native code can take substantial time. If the partitioning of tasks between the foreign and the host languages is not used properly, this can cause a dispersion of the responsibility to perform a simple task between several languages. In some cases we can have excessive calls and passage of parameters between one language to another, These calls add more complexity to the program and negatively impact the performance of the system[1].

- **Refactoring: To refactor this design smell, start by locating the classes and objects involving excessive communication. Identify related attributes and operations. Then try to split the responsibility in a way that minimises the calls between the different languages but also with considering high cohesion.** Decide which tasks are better implemented in which language. A good solution would be to separate the responsibility and identify the common concerns. If needed isolate the module involving the excessive calls or provide a wrapper to minimise the calls when we need to access from one language features available in another language.

- **Benefits of the Refactoring:** Refactor this design smell ensure high cohesion and low coupling. A better performance by reducing the number of calls from one language to another. It also reduces the complexity, by limiting and splitting the responsibility

between the host and the foreign code. Another benefit is to avoid unnecessary broken code related to a non separation of concerns when applying changes.

- **Related Design Smells:** Circular dependency.
- **Related Patterns:** Message Redirector [81] and Adapter [44].
- **Examples:** This design smell can be illustrated when in a system where we have different calls from one language to another. Occurrences of this design smell are generally observed in systems involving different layers or components. For example, the same object can be used and-or modified in more than one modules written in different languages. Each time we need the object, we pass it from one language to another or we call the foreign method to perform specific tasks in the object. The solution would be to separate the responsibilities and minimise the calls between the languages. It is better to focus each time in a single language to implement the tasks. Some examples of this design smell have been observed in *Godot, PortAudio Java Bindings, OpenResty*. In *Godot*, The function *process()* is called at each time delta. The time delta is a small period of time that the game does not process anything *i.e.,* the engine does other things than game logic out of this time range. The foreign function *process()* is called multiple times per second, in this case once per frame[5]. Another example in *PortAudio Java Bindings*, where they used raw buffers between Java and C++. In this example, they copy data between buffers which makes way for more communication than needed. Java supports memory mapped input/output for this purpose, with this raw buffers can be used between language barriers[6]. We present in Fig. 7.2, an example of occurrences of this design smell extracted from *OpenResty*. We found a situation, in which a developer introduced several calls from one Lua file to the Nginx. In this example, the developer was excessively calling the function ngx.exec() from the Lua file and getting values from the configuration file. The good solution is also present in the same system. As they usually provide access as an entry point to ensure the better way of communication between Nginx and Lua[7].

Too Much Clustering

- **Overview:** Too many native methods declared in a single class would decrease readability and maintainability of the code. This will increase the lines of code within that

[5] `https://github.com/godotengine/godot-demo-projects/blob/master/2d/pong/paddle.gd`
[6] `https://github.com/rjeschke/jpa/blob/master/src/main/native/jpa.c#L84`
[7] `https://github.com/openresty/lua-nginx-module`

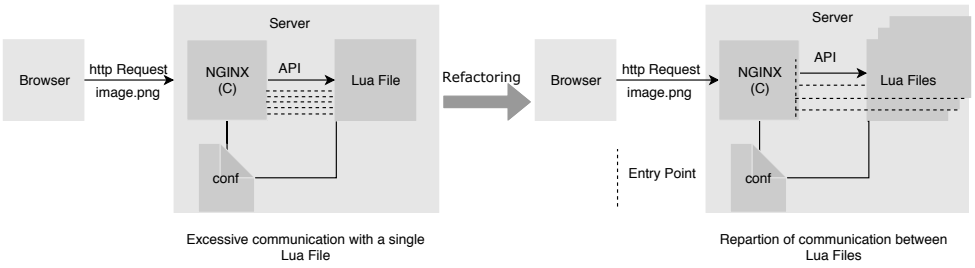


Figure 7.2 Illustration design smell - Excessive Inter-language Communication

class and thus make the code review process harder. Many studies discussed good practices about the number of methods to have within the same class, some examples are the rule of 30 introduced by Martin Lippert [121], or the *7 plus/minus 2 rule* stating that a human mind can hold and comprehend from five to 9 objects. Most of the relevant measures are the coupling, cohesion, the single principle responsibility, and the separation of concerns. In this context, a bad practice would be to concentrate multi-language code in few classes, regardless of their role and responsibilities.

- **Context:** In a situation where we are developing a new system or a system which has been released and we are asked to add new features. We are considering that the case of a multi-language system. The features to add may be in the same foreign language but are not related to each other. Each one of them is related to a specific task.

- **Problem: The problem is that under pressure, developers may excessively try to limit the classes participating in the multi-language programming which may violate the separating of concerns principle**. This may be related to concerns in term of tasks as well as concerns in term of programming languages. Multi-language code is difficult to maintain and understand, having multi-language code scattered through the project may negatively impact the maintenance activities. For that, developers may choose to always limit the classes containing the multi-language code.

- **Bad Solution:** The bad solution would be to always try to concentrate as much as possible the multi-language code in the same classes without considering the responsibilities related to each class. This situation can also be defined by the merge the multi-language code in a single class in a way that results in a high coupling and low cohesion. The occurrence of this design smell would appear if we do not consider the concerns when adding new features or functionalities that involve the use of a multi-

language code. The allocation of responsibilities between the multi-language classes is not well managed during system evolution so that one module becomes predominant regarding the other modules.

- **Consequences of the Design Smell:** As a consequence of this design smell, would be a negative impact on maintainability, as applying a change would require an important effort due to the complexity of understanding such code. There is also a loose of portability and reusability as the module has more than one responsibility. If we do not consider the cohesion and concerns when adding the code, this can result in a high coupling with low cohesion.

- **Refactoring:** To refactor this design smell, **start by identifying and grouping related attributes and operations in term of concerns. Then, search or create classes that could host these attributes and operations and ensure a high cohesion. Then eliminate unnecessary coupling and indirect associations to have a high cohesion with low coupling. We encourage decoupling the code into distinct units with well-defined responsibilities**. Always separate the concerns. When the concerns are properly separated, we can have different teams working in parallel on a given feature. A component with a solid separation of concerns can ensure greater collaboration between developers, maintainers, designers, etc. They can work at the same time on the same component. We also recommend ensuring cohesion between the programming languages and not only a cohesion of responsibilities. Depending on the programming languages, a possible solution would also be to expose services of a specific language and use extensions to invoke each programming language.

- **Benefits of the Refactoring:** Refactor this design smell will introduce several benefits, including the separation of the concerns and having simple and readable classes. This can also reduce maintainability efforts by keeping classes cleaned. Another benefit would be to allow high cohesion and low coupling.

- **Related Design Smells:** Too Much Scattering of Multi-language Participants, Blob, and Swiss Army Knife [27].

- **Related Patterns:** Interface Extension [122].

- **Examples:** This design smell can be identified in multi-language systems when the multi-language code is mixed in the same classes or files without any common concerns.We believe that it is a good practice to not spread the multi-language code through

the system, but this should be balanced regarding the concerns. A good solution would be to find a compromise between separating the concerns and not dispersing the multi-language code. When a change needs to be applied we should be able to easily locate the code directly associated with the change. If the concerns are well separated between the languages, it is easier for developers to work separately in different tasks or modules. Separating the concerns also help to avoid breakage in unrelated features, if the features are mixed together a change to the behavior of one may cause a bug in another feature. An example is *React*, as it was reported to violating the separating of concerns by mixing *JavaScript* code with *HTML*, and *CSS*[8]. We present in Fig. 7.3 an example extracted from *ZMQ JNI*[9]. In this example, native methods related to cryptographic operations are mixed in the same class as the methods used for network communication. This merging of concerns resulted in a blob multi-language class that contains 29 native declaration methods and 78 attributes.Another example, the class GodotLib which contains 25 native declaration methods[10].

Too Much Scattering

- **Overview:** Similar to too much clustering, when using multi-language code, developers and managers often have to decide on a trade-off between isolating or splitting the native code. Accessing this trade-off is estimated to improve the readability and maintainability of the systems [49]. This design smell occurs when classes are scarcely used in multi-language communication without satisfying both the coupling and the cohesion.

- **Context:** We are maintaining a system, migrating a project from one language to another, or adding new functionality and features available in other languages. In multi-language systems usually, several teams are involved in the same project. This can also be faced in microservices architecture and feature-based decomposition where several teams are working in different features.

- **Problem: Under time pressure developers wants to add multi-language systems code, the problem is that developers in these situations are not always sure where they should add the code**. Especially that developers or maintainers

[8] `http://krasimirtsonev.com/blog/article/react-separation-of-concerns`

[9] `https://github.com/zeromq/zmq-jni/blob/master/src/main/java/org/zeromq/jni/ZMQ.java`

[10] `https://github.com/godotengine/godot/blob/60d910b1916305c4b0ac5f92415083995b4f7c7a/platform/android/java/src/org/godotengine/godot/GodotLib.javanativemethods`

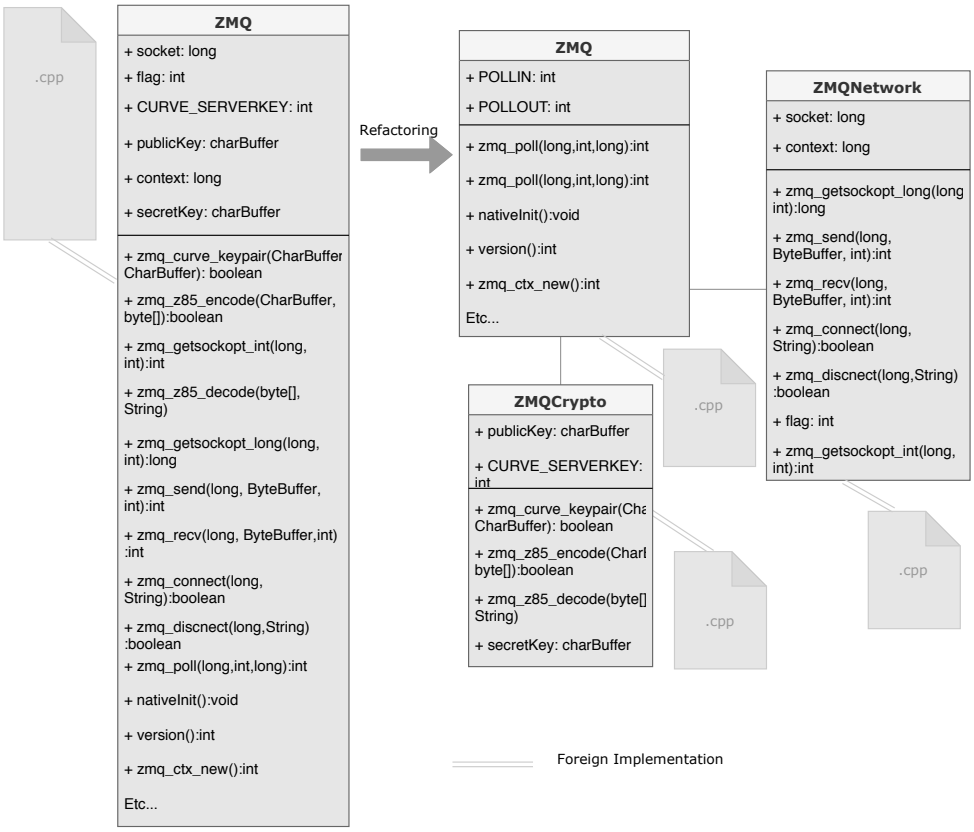


Figure 7.3 Illustration design smell - Too Much Clustering

do not always have a global idea about the overall architecture and design of the system. When several developers or teams are involved in the same project bugs related to changes may occur. Developers and managers would avoid these breakages in unrelated features, if the features are mixed together a change to the behavior of one may cause a bug in another feature.

- **Bad Solution:** Try to always separate multi-language classes to avoid breakage in unrelated features without considering the concerns. Add the foreign code without considering the concerns and architecture of the project. Each time we estimate that the use of multi-language programming can be easier to perform a specific task, we add the foreign code without considering the classes already participating in the multi-language code and the responsibilities of each class.

- **Consequences of the Design Smell:** The methods and classes participating in the foreign interaction are spread through the code in a way that determining which classes are participating and which does require some effort. This code will be more difficult

to maintain and refactor. It would be hard to know which classes are participating in the multi-language programming and which are not. It becomes difficult to locate and fix issues related to multi-language programming.

- **Refactoring:** To refactor this design smell, start by investigating the architecture of the project, which classes and packages are better involved in the multi-language programming concept. Then identify the multi-language code (*e.g.,* methods, attributes, etc.) that are scattered through the code and that could be grouped in term of concerns. Once the above located, **try to isolate the foreign code and limit the number of classes participating in the multi-language programming. Such Classes should be easily located in both of the languages, so they can easily be refactored or modified**. It is better to concentrate the code participating in the multi-language programming, so we have classes with and classes without.

- **Benefits of the Refactoring:** When applying a change, developers or maintainers can easily locate the code related to the same feature. The refactored solution will ensure high cohesion and low coupling. Another benefit is to isolate the foreign code and limit the number of multi-language classes.

- **Related Design Smells:** Functional Decomposition [27].

- **Related Patterns:** Component Wrapper [14], Interface Extension [122], Facade, and Decorator [44].

- **Examples:** This design smell can be observed in a system where we have many classes participating in the multi-language programming and most of them contain only a small part involving foreign code as illustrated in Fig. 7.4. These classes are mainly mono-language but contain few foreign codes. A good solution would be to refactor the code and isolate the foreign code in a way that some classes are mainly participating in the multi-language programming and others involve only one language. We present a simple example in Fig. 7.4 to illustrate the excess of classes participating in multi-language programming. In this example, we have three classes each of them contains two native methods declaration. A good solution would be to move these methods or add a superclass if needed, that will contain all the native declaration methods, and keep these classes as inherited from this superclass. This will reduce the number of native method declaration by removing the duplicated ones. This will also reduce the scattering of multi-language participants and concerns by keeping the multi-language code concentrated only in specific classes. In the same vein, in the system *jpostal*, the

classes *AddressParser* and *AddressExpander* contain each few native declaration methods that could be grouped into the same class[11]. Especially that the implementation of most of these native methods is duplicated between both of them. Other classes also from the same package contain one to two native method declaration. Another example of this design smell is present in *Frostwire*. For example, the method *getWindowHandleNative()* is the only function written in C, and the window handle is used for displaying video using mplayer[12]. This method could have been grouped with other natives methods to reduce the number of classes participating in the multi-language code. There are also other ways of doing this in Java by using a video player made for Java.

Passing Excessive Objects

- **Overview:** Accessing field's elements by passing the whole object is a common practice in object oriented programming. However, in the context of JNI, since the Object type does not exist in C programs, passing excessive objects could lead to extra overhead to properly perform the type conversion. Indeed, this design smells occurs when developers pass a whole object as an argument, although only some of its fields were needed, and it would have been better for the system performance to pass only those fields except the purpose to pass the object to the native side was to set its elements by the native code using *SetxField* methods, with x the type of the field. Indeed, in the context of object-oriented programming, a good solution would be to pass the object offering a better encapsulation, however, in the context of JNI, the native code must reach back into the JVM through many calls to get the value of each field adding extra overhead. This also increases the lines of code which may impact the readability of the code [49].

- **Context:** We have some attributes from classes and objects in the host language that we must access and use in the foreign code.

- **Problem: Developers do not have enough knowledge about the performance cost when integrating several programming languages.** They usually take design and coding decisions considering a single paradigm and do not consider that combining distinct paradigms may change those decisions.

[11] `https://github.com/openvenues/jpostal/tree/master/src/main/java/com/mapzen/jpostal`

[12] `https://github.com/frostwire/frostwire/blob/7414e3be2ef5ced88a775df7831b7ae382fcf966/desktop/lib/native-src/linux/SystemUtilities.cpp`

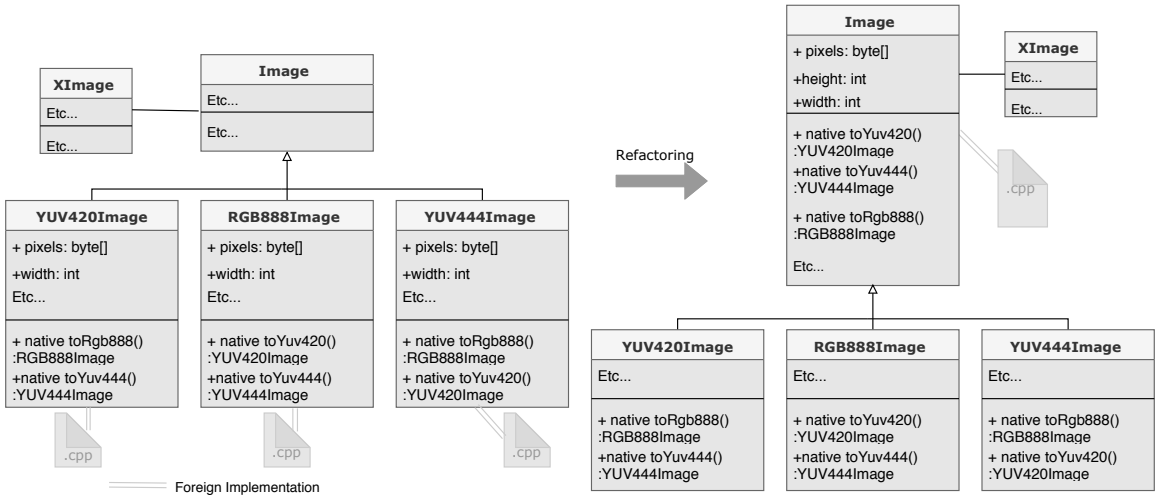


Figure 7.4 Illustration design smell - Too Much Scattering

- **Bad Solution:** We usually have to decide whether we pass an object that has multiple fields or the fields individually. The bad solution would be to always favor passing a whole object instead of passing parameters; *i.e.,* each time we pass the whole object instead of passing the parameters of interest. If we consider passing the whole object in the context of object-oriented principle, this provides better encapsulation. However, in the case of multi-language systems, it is better to consider the performance cost between the two solutions when combining different paradigms and languages.

- **Consequences of the Design Smell:** Passing an object from one language to another may require an important effort of performance and implicate intermediate methods to access the native code as not all the programming languages have or treat similarly the types. In some cases, the native code uses several foreign calls to get the value of each individual field. Such additional calls add extra costs. Calls from native code to host language code is more expensive than a normal method call and may negatively impact the performance[1]. Other consequences are that the methods implicated by this design smell will not have many parameters and will favor the encapsulation.

- **Refactoring:** To remove this design smell, **a good solution would be when few parameters are needed to be accessed, favor passing them separately instead of passing the whole object.** Depending on the languages, it may require additional effort to access the fields if they are not passed as parameters.

- **Benefits of the Refactoring:** This will improve the performance in the case where passing a whole object is a consuming task. It also improves the readability by hav-

ing the parameters of interest instead of whole objects. Another benefit is to avoid calling heavy methods to extract the parameters from the object, especially when the programming languages differ in term of types and paradigms.

- **Examples:** An example of occurrences of this design smell has been discussed in *IBM website*[1]. In the case of JNI, when we pass objects, it results in many calls to get the value for each of the individual fields. This kind of calls add an extra cost as the interactions between the native code and the Java code is generally more expensive than a method call. It may negatively impact performance. Figure 7.5 presents an example of occurrences of this design smell. While figure 7.6 presents a possible refactoring to remove this design smell. Depending on the programming language this design smell may also occur in Python/C and other sets of programming languages.

```
/* C++ */
int sumValues (JNIEnv* env,jobject obj,jobject allVal)
{ jint avalue= (*env)->GetIntField(env,allVal,a);
  jint bvalue= (*env)->GetIntField(env,allVal,b);
  jint cvalue= (*env)->GetIntField(env,allVal,c);
  return avalue + bvalue + cvalue;}
```

Figure 7.5 Design smell - Passing Excessive Objects

```
/* C++ */
 int sumValues (JNIEnv* env, jobject obj, jint a, jint b,jint c){ return a + b +
     c;}
```

Figure 7.6 Refactoring - Passing Excessive Objects

Unused Parameters

- **Overview:** Long list of parameters make methods hard to understand [123]. It could also be a sign that the method is doing too much or that some of the parameters are no longer used. In the context of multi-language programming, some parameters may be present in the method signature however they are no longer used in the other components written in different programming languages. Since multi-language systems usually involve developers from different teams, those developers often prefer not to remove such parameters because they may not be sure if the parameters are used by other components.

- **Context:** When adding new features or modifying an existing project, it may happen that we are not sure which parameters to keep and which one to remove. This can also happen when passing parameters to and from one language to another which were never been used in the other language.

- **Problem:** Several teams and developers are involved in the same projects. **These projects are then maintained by other developers that do not have enough knowledge about the architecture of the project**.

- **Bad Solution:** A bad solution would be, when applying a change to always keep the parameters already existing as they may be used in the other language while they are no longer used. This can also appear when we pass all the parameters that we believe can be used to complete the task while concretely not all of them are used.

- **Consequences of the Design Smell:** Having unused parameters from one language to another may add complexity to the code especially in the maintenance activities. Developers may not be sure which parameters should be used and which are not as they are related to another language. Multi-language systems are by nature more difficult to understand, adding unnecessary parameters or applying a change and not removing the corresponding parameters will introduce more complexity to the system. Some developers may go through this solution as once all the parameters are defined and passed from one language to the other, it is easier to use them or apply changes that involve these parameters.

- **Refactoring:** To remove this design smell, **Keep only the parameters that are used to avoid introducing unnecessary complexity and improve the readability.**

- **Benefits of the Refactoring:** Improve the understandability and maintainability as the method will contain only the parameters used. This also avoids dead code and Keep only the parameters needed.

- **Examples:** Figure 7.7 presents an example of occurrences of this design smell. The parameter *acceleration* is defined in the native method signature. However, it is not used by the native code. The solution would be to remove the unused parameters.

```
/* C++ */
JNIEXPORT jfloat JNICALL Java_jni_distance
  (JNIEnv *env, jobject thisObject,
  jfloat time, jfloat speed,
  jfloat acceleration) {
    return time * speed;}
```

Figure 7.7 Design smell - Unnecessary Parameters

Unused Native Method Implementation

- **Overview:** This appears when a method is declared in the host language (Java in our case) and implemented in the foreign language (C or C++). However, this method is never called from the host language. This could be a consequence of migration or refactoring in which developers opted for keeping those methods to not break any related features.

- **Context:** We have the method declaration and its corresponding implementation. However, it is never called from the host language. In the case of multi-language programming, it is hard for a developer working on a specific part of the project implemented in a single language, to know which methods are really used in the other language. Some implementations could also be provided by different Dynamic Link Library not written in the same language. These systems usually involve several developers or teams to work separately in the project and access only a sub-part of it.

- **Problem: Several developers working on the same code and maintainers do not have enough knowledge about the project to correctly identify whether the code is used or not.** It can also be in situations where a project was migrated or refactored. This can also be related to a planned extension that never happened or renaming that failed. In the case of multi-language systems, it can be more difficult to locate these methods as they are implemented in a language or component and used in another one. Developers should have a complete vision of the architecture of the systems to know which methods are used or are planned to be used in near future releases. Depending on the programming language and paradigm, we may face situations where the foreign method is not called using the same name as the one used in the implementation or with the same signature.

- **Bad Solution:** Always keep the native methods implementations without additional

checks as they may be used in the other language. Avoid breakages related removing code that is still called or used somewhere on the project.

- **Consequences of the Design Smell:** If a future modification involves using these methods, it will be easier as they are already implemented. However, this design smell adds more complexity and may result in huge classes in which we have implementation of methods that are never called from the other language. When fixing bugs or adding new features, the developers may go through these methods and will not be aware that they are not really used.

- **Refactoring:** To remove this design smell, **remove all unnecessary and unused code to reduce the complexity and keep in each class only the methods that are really used.** To prevent occurrences of this design smell, it is also important to always remove all the code related to the multi-language programming if it is no longer used.

- **Benefits of the Refactoring:** Improve the understandability and maintainability as the code will contain only the methods that are used. This also avoids dead code by having clean code and Keeping only the methods used. It may also be easier for a maintainer or new developer to locate the code used.

- **Examples:** An example of this design smell was initially perceived when we manually analyzed JNI systems and found some native methods that have been declared and implemented but are never called [120]. It may be due to changes or refactoring in which they introduced another method. These methods introduced some doubt as we were confused where they were used, but then we semi-automatically checked if they were called using *grep* command but we did not find any calls to these methods.

Unused Native Methods Declaration

- **Overview:** Similar to *Unused Method Implementation*, this design smell (also known as *Missing Implementation*) occurs when a method is declared in the host language but is never implemented in the native code. This smell and the previous one are quite similar. However, they differ in the implementation part, while for the smell *Unused Method Implementation*, the method is implemented but never called, in case of the smell *Unused Method Declaration*, the unused method is not implemented and never called in the foreign language. Such methods could remain in the system for a long period of time without being removed because having them will not introduce any bug

when executing the program but they may negatively impact the maintenance activities and effort needed when maintaining those classes.

- **Context:** When we have some methods declaration in the host language that has never been implemented in the foreign language.

- **Problem: Requirement or functionalities changes may lead to unused code.** Usually, different teams may be involved separately to contribute in each programming language. These teams do not have a global view of the whole system, which methods are used and which are not.

- **Bad Solution:** A bad solution would be when applying a change to always keep the native methods declared without additional checking as they may be used in the other language while they are no longer used.

- **Consequences of the Design Smell:** If a future modification involves implementing these methods, it will be easier as they are already declared. However, this design smell can result in unused and unnecessary code. It may add some complexity to the code and introduce more difficulty when reading and maintaining the code. Depending on the languages, this kind of methods may not crash the system or display an error, as these methods are never called or used. However, for a maintainer, it would require additional effort to investigate which methods are really used in the multi-language systems and which are not.

- **Refactoring:** To remove this design smell, **keep only the methods that are used in the multi-language systems' interaction.** An unused code may negatively impact the quality of a system, the impact may be important when we are dealing with multi-language systems. Depending on the size of the system, it may be difficult for a maintainer to identify the methods used. To retrace or fix a bug this may require more effort.

- **Benefits of the Refactoring:** Improve the understandability and maintainability as the code will contain only the methods that are used. This also avoids dead code by providing clean code and Keeping only the methods used. Another benefit is that it would be easier for a maintainer or new developer to locate the code used.

- **Examples:** An example of this design smell has been perceived when we analyzed JNI systems and collected the number of method implementation and the number of

a method declaration [120]. In most of the system, the number was the same between both of these metrics. However, we found examples where some native methods have been declared but have never been used.

Not Handling Exceptions Across Languages

- **Overview:** The exception handling flow may differ from one programming language to the other. In case of JNI systems, developers should explicitly implement the exception handling flow after an exception has occurred[1] [11,61,63]. Since JNI exception does not disrupt the control flow until the native method returns, mishandling JNI exceptions may lead to vulnerabilities and leave security breaches open to malicious code [11,61,63].

- **Context:** In the case of multi-language systems, depending on the language we may not have the same way to manage the exception.

- **Problem: The management of exceptions is not automatically ensured in all the languages.** Some programming languages, require developers to explicitly implement the exception handling flow after an exception has occurred. If the exception is not explicitly implemented and handled by the developer this may introduce bugs. Developers may also not be aware of the consequences of not managing the exceptions, especially in the case of multi-language programming.

- **Bad Solution:** The bad solution would be to always rely on the exception provided by the other language and not necessarily implement the exception handling.

- **Consequences of the Design Smell:** If the exception is not explicitly implemented and handled by the developer. This may result in bugs and unchecked exceptions will introduce bugs in the system that will be hardly debugged or retraced to the origin of the bug.

- **Refactoring:** To remove this design smell, **always check whether an exception has been thrown after invoking any foreign methods that may throw an exception.** Multi-language systems introduce more complexity than mono-language systems and need more effort to fix bug and issues, it is important to consider checking and handling exceptions to prevent issues related to not checking exception. In the case of multi-language systems, it is much easier to prevent crashes by implementing the exception than to debug after the crash occurred. Upon handling the exception,

we should also clear it depending on the language. For JNI, we should use the *ExceptionClear* function to inform the JVM that the exception is handled and JNI can resume serving requests to Java space. If the host language provides the handling and management of exceptions, it possible to simply check if an exception has occurred in the foreign code and if so return immediately to the host code so that the exception is thrown. It will then be either handled or displayed using the exception-handling process provided by the host language.

- **Benefits of the Refactoring:** The refactored solution introduces several benefits, including: prevent crashes, separate error-handling code from regular code, and differentiating error types.

- **Examples:** Examples of occurrences of this design smell have been discussed in developers' documentation as a wise practice[1] [13]. Most of the systems that we analyzed were not always implementing a proper way to handle the exception as shown in Fig. 7.8, this code may cause a crash if charField field no longer exists. For the JNI case, one good example was *Libgdx*, where they catch Java exceptions in native code using the JNI API call ExceptionOccurred. Figure 7.9 presents a refactoring example extracted from *IBM Developer Site*[1]. Occurrences of this design smell will not block the execution of the native code. However, any calls to JNI API will silently fail. As the actual exception does not leave any traces behind, it is hard to debug.

```
/* C++ */
jclass objectClass;
jfieldID fieldID;
jchar result = 0;
objectClass= (*env)->GetObjectClass(env, obj);
fieldID= (*env)->GetFieldID(env, objectClass, "charField", "C");
result= (*env)->GetCharField(env, obj, fieldID);
```

Figure 7.8 Design smell - Not Handling Exceptions Across Languages

Assuming Safe Multi-language Return Values

- **Overview:** Similar to the previous design smell, in the context of JNI systems, not checking return values may lead to errors and security issues [49,61]. The return values

[13] https://nachtimwald.com/2017/07/09/jni-is-not-your-friend/

```
/* C++ */
jclass objectClass;
jfieldID fieldID;
jchar result = 0;
objectClass = (*env)->GetObjectClass(env, obj);
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
if((*env)->ExceptionOccurred(env)) {return;}
result = (*env)->GetCharField(env, obj, fieldID);
```

Figure 7.9 Refactoring - Not Handling Exceptions Across Languages

from JNI methods indicates whether the call succeeded or not. It is the developers' responsibility to always perform a check before returning a variable from the native code to the host code to know whether the method ran correctly or not.

- **Context:** Typically when we are implementing a multi-language system, we need to access and transfer data and information between different languages. We usually pass and return values from one language to another.

- **Problem:** Exceptions are extensions of the programming language for developers to report and handle exceptional events that require special processing outside the actual flow of the application. However, the management of exception is not supported by all the languages. The same for return values that are used to transfer data from one language to another. **Some developers assume that return values are safe, others are not aware of the consequences of not checking multi-language return values.**

- **Bad Solution:** We may need to implement a specific task in a certain language and need to have the value returned to the other language. In most of the cases, we are just returning the value without performing specific checks. The bad solution in case of multi-language systems is to implement the method in the foreign language and have its result returned to the main language assuming that return values are safe without considering additional checks.

- **Consequences of the Design Smell:** It is important to consider the return values as exceptions to verify that the interaction between the languages was well performed. Otherwise, it may result in introducing faults and bugs in the program. As some values may be wrong or simply empty which can cause problems when returned to the other language.

- **Refactoring:** To remove this design smell, **a good solution would be to never assume that it is safe to use a value returned by a language API call, which must always be checked to make sure that the call was successfully executed and the proper usable value is returned to the native function**. Multi-language methods usually have a return value that indicates whether the call succeeded or failed. API functions rely on their return values to indicate any errors during the execution of the API call.

- **Benefits of the Refactoring:** Ensure that the interaction between the languages was well performed. Other benefits are to ensure that the usable value is returned to the foreign code and avoid introducing bugs.

- **Examples:** Examples of occurrences of this design smell have been observed in most of the open source systems that we analyzed. This was also reported in several developers' documentation and bug reports[14]. Depending on the languages involved in the multi-language system, it is mostly recommended to always check the return values from one language to another. As in most of the cases, we use another language to perform a calculation or specific features that will be then used by the main language. It is recommended to always check the value before returning it to the host language. As illustrated in Fig. 7.10 extracted from *Libgdx*, if the class *NIOAccess* or one of its methods is not found, the native code will cause a crash. As we are not applying any check to handle the problems related to the return values. A good solution to remove this design smell is to add a check that handles the situations in which problems may occur with the return values. Figure 7.11 presents a good example to illustrate a possible refactored solution.

```
/* C++ */
staticvoid nativeClassInitBuffer(JNIEnv *_env){
 jclass nioAccessClassLocal= _env->FindClass("java/nio/NIOAccess");
 nioAccessClass=(jclass) _env->NewGlobalRef(nioAccessClassLocal);
 bufferClass=(jclass) _env->NewGlobalRef(bufferClassLocal);
 positionID= _env->GetFieldID(bufferClass, "position", "I");
```

Figure 7.10 Design smell - Assuming Safe Multi-language Return Values

[14] https://www.developer.com/java/data/exception-handling-in-jni.html

```
/* C++ */
//Checking the Return Value of JNI API Calls
jclass clazz;
...
clazz = env->FindClass("java/lang/String");
if (0 == clazz) {/* Class could not be found. */
} else { /* Class is found, we can use the return %value.*/ }
```

Figure 7.11 Refactoring - Assuming Safe Multi-language Return Values

Not Caching Objects' Elements

- **Overview:** To access Java objects' fields from native code through JNI and invoke their methods, the native code must perform calls to predefined functions *i.e., FindClass(), GetFieldId(), GetMethodId(), and GetStaticMethodId()*. For a given class, IDs returned by *GetFieldId(), GetMethodId(), and GetStaticMethodId()* remain the same during the lifetime of the JVM process. The call of these methods is quite expensive as it can require significant work in the JVM. In such situation, it is recommended for a given class to look up the IDs once and then reuse them. In the same context, looking up class objects can be expensive, a good practice is to globally cache commonly used classes, field IDs, and method IDs.

- **Context:** When implementing a multi-language system, we need to pass objects and variables from one language to the other. In this case, we want to access an object's field and methods from the foreign code.

- **Problem: Developers do not have enough knowledge of how the fields and methods are retrieved when passing from one language to another.** They may consider using the most simple way to access foreign code and do not consider the performance cost.

- **Bad Solution:** Depending on the language, use the available methods to access the objects fields and methods. Each time we need one of the object's field, call the methods to retrieve the field as if it is the first time we access the field. As example to access Java objects' fields and their methods, the native code perform calls to FindClass(), GetFieldID(), GetMethodId(), and GetStaticMethodID().

- **Consequences of the Design Smell:** Depending on the language, it may require an important effort to use available methods to access the object fields and methods.

Although these methods may be used frequently in multi-language systems, they may be heavy function calls by their nature. These functions traverse the entire inheritance chain for the class to identify the ID to return. The IDs returned for a class using GetFieldID(), GetMethodID(), and GetStaticMethodID(), do not change during the lifetime of the JVM process. However, this may be expensive in term of performance. For that, we recommend to look them and reuse them once needed.

- **Refactoring:** Neither the Class object, the Class inheritance, nor the fieldID can be changed during the execution of the system. These values are cached in the native layer for subsequent accesses. The return type of the FindClass function is a local reference, so to cache its values, developers must create a global reference first through the NewGlobalRef function when it is needed. The return value of GetFieldID is jfieldID, which is an integer that can be cached as it is. To remove this design smell, **developers should focus on caching both the field and method IDs that are accessed multiple times during the execution of the application,** this practice makes an improvement in the execution time.

- **Benefits of the Refactoring:** The IDs are often pointers to internal runtime data structures. Looking them up may require several string comparisons. Once we have them the call to get the field or the method does not take an important time. This also improves performance by avoiding several lockups and avoid calling heavy functions.

- **Examples:** Examples of occurrences of this design smell have been observed in JNI systems. Some developers' documentation also reported this common bad practice as negatively impacting the performance as shown in Fig. 7.12[1]. In the case of JNI, a correct way to initialise the IDs is to Create a method in the C/C++ code that performs the ID lookups. The code will be executed once when the class is initialised. If the class is unloaded and then reloaded, it will be executed again. If commonly used classes, fields Ids, and methods Ids are not properly cached, we lose the benefit of using the C/C++. This design smell negatively impacts the performance. As presented in the example[1], using caching field IDs will take 3,572 ms to run 10,000,000 times 7.12. However, without using the cache as illustrated in 7.13, it takes 86,217 ms. Using this design smell the task takes 24 times longer than without the occurrences of this design smell.

```
/* C++ */
int sumVal (JNIEnv* env,jobject obj,jobject allVal){
   jclass cls=(*env)->GetObjectClass(env,allVal);
   jfieldID a=(*env)->GetFieldID(env,cls,"a","I");
   jfieldID b=(*env)->GetFieldID(env,cls,"b","I");
   jfieldID c=(*env)->GetFieldID(env,cls,"c","I");
   jint aval=(*env)->GetIntField(env,allVal,a);
   jint bval=(*env)->GetIntField(env,allVal,b);
   jint cval=(*env)->GetIntField(env,allVal,c);
   return aval + bval + cval;}
```

Figure 7.12 Design smell - Not Caching Objects' Elements

```
/* C++ */
   jint aval=(*env)->GetIntField(env,allVal,a);
   jint bval=(*env)->GetIntField(env,allVal,b);
   jint cval=(*env)->GetIntField(env,allVal,c);
   return aval + bval + cval;
```

Figure 7.13 Refactoring - Not Caching Objects' Elements

Not Securing Libraries

- **Overview:** A common way to load the native library in JNI is the use of the method *loadLibrary* without the use of a secure block. In such situation, the code loads a foreign library without any security check or restriction. However, after loading the library, malicious code can call native methods from the library, this may impact the security and reliability of the system [49, 124].

- **Context:** We want to access foreign libraries or an API available in another language. We aim to integrate an external library with the main application developed in a different programming language.

- **Problem: Developers are not always aware of the consequences of insecure code** or do not provide enough intention.

- **Bad Solution:** When developing multi-language systems, we always need to access some API or libraries implemented in another language. We load the native library or API directly in the code without any security checks or restriction.

- **Consequences of the Design Smell:** As consequences of the occurrence of this design smell, several problems may occur due to the leak of security. An unauthorised code may access and load the libraries. Malicious code may use this vulnerable code to access the system. Depending on the domain of application in which the multi-language development has been involved, this may have an important impact. As for mobile application or embedded systems, a fault in the security may have an impact at the human level.

- **Refactoring:** To remove this design smell, **always ensure that the libraries cannot be loaded without permissions.** It is important to ensure that the loading of external libraries is written in a secured block of code to guarantee access only to those who are allowed to. Depending on the language some predefined classes may ensure security and prevent undesirable access to the system. As for the Java language, it is recommended to always load libraries in static blocks, wrapped in a call to *AccessController.doPrivileged* or use the *securityManager*.

- **Benefits of the Refactoring:** One of the main benefits is to ensure that the libraries cannot be loaded without permissions. This also avoids malicious attacks and secure the load of the library and the project.

- **Examples:** Occurrences of this design smell have been observed on most of the analyzed systems. In the JNI case, we found the usage of the secure library only with the *JDK* and *Openj9*. In these systems, the loading library is always performed in a static block and using the *AccessController*. *AccessController* presents a safe way to load a library because it ensures that the library cannot be loaded without permissions, as shown in Fig. 7.14. Depending on the languages, we recommend to always secure the loading library by using available methods for the specific language.

```
/* Java */
 static { AccessController.doPrivileged(
         new PrivilegedAction<Void>() {
         public Void run() {
         System.loadLibrary("osxsecurity");
         return null; } } ); }
```

Figure 7.14 Securing Library Loading

Hard Coding Libraries

- **Overview:** Let us consider a situation in which we have the same code to run on different platforms. We need to customize the loading according to the operating system. However, when those libraries are not loaded considering operating system (OS) specific conditions and requirements, but for instance with hard coded names and a try-catch mechanism, it is hard to know which library has really been loaded which could bring confusion especially during the maintenance tasks.

- **Context:** We are loading different libraries for different OS, the same code can not run in all the platforms. we need to customise the loading according to the OS. For that, we hard-code the loading according to the OS.

- **Problem:** Project may be designed as a prototype or do not consider future extensions and adaption to new platforms.

- **Bad Solution:** Depending on the used language, some of them are expected to run on all platforms, but in other languages, there must be different native code libraries for different platforms, which must be loaded according to the target OS. To ensure loading the libraries according to the OS the loading libraries is hard-coded in the code.

- **Consequences of the Design Smell:** When the libraries are hard-coded, it is difficult for a maintainer to know which library is loaded in which time. Even to handle bugs and errors this would require more time to locate the errors. As consequences of this design smell, developers may require additional effort to distinguish between the different libraries. This may impact the understandability and readability of the system.

- **Refactoring:** To remove this design smell, **a clean way to load the library would be to handle all targeted OS on which the library is available.** This ensures better code readability, letting the code Reader directly knows what libraries are being loaded. Also, loading in a way to take care of the OS makes sure that all cases are properly covered and if a code is running on a new OS, errors are easy to locate.

- **Benefits of the Refactoring:** One of the main benefits is to ensure readability by making the libraries easily defined for each operating system. It also ensures handling all targeted OS on which the library is available and improve the understandability.

- **Examples:** Examples of occurrences of this design smell as well as the good solution has been observed respectively in `JavaSmt` and `Frostwire`. In `JavaSmt`, most of the

loading libraries were hard-coded in a way that it was difficult for us to know which library is related to which OS. As shown in Fig. 7.15. Some of the comments were explaining the OS related to the library. However, it is better if the way to load the library can be self-efficient and reflect which library is loaded. It is important when loading libraries to take care of the OS as shown in Fig. 7.16. This ensures that all platforms are covered and those missing libraries can be easily identified.

```
/* Java */
public static synchronized Z3SolverContext create(
try { System.loadLibrary("z3"); System.loadLibrary("z3java");
} catch (UnsatisfiedLinkError e1) {
try { System.loadLibrary("libz3");
      System.loadLibrary("libz3java");
} catch (UnsatisfiedLinkError e2) {...}
```

Figure 7.15 Design smells - Hard Coding Libraries

```
/* Java */
/*for Windows*/
if (OSUtils.isWindows() && OSUtils.isGoodWindows()) {
if (OSUtils.isMachineX64()) {
System.loadLibrary("SystemUtilitiesX64";}
else { System.loadLibrary("SystemUtilities");}
/*for Mac OS*/
public final class GURLHandler {
System.loadLibrary("GURLLeopard");
public class MacOSXUtils {
System.loadLibrary("MacOSXUtilsLeopard");
```

Figure 7.16 Refactoring - Hard Coding Libraries

Not Using Relative Path to Load the Library

- **Overview:** This smell occurs when the library is loaded by using its name instead of the corresponding specifying the path. Using a relative path, the native library can be loaded and installed everywhere. However, the use of an absolute library path can introduce future bugs in case the library is no longer used. This may also impact the reusability of the code and its maintenance because the library can become inaccessible due to incorrect path.

- **Context:** When implementing a multi-language system, we need to load foreign code and then use external libraries or API. We have to specify the name of the library that we are going to load.

- **Problem:** The project was designed as a prototype not for future reuse. This can also be related to a situation where **the project was initially used locally by a single or few developers.**

- **Bad Solution:** In multi-language systems we usually need to access or integrate foreign libraries or API. For that, we need to specify the name or path to access the library. A bad solution would be to load the external library by only specifying the name without providing the full path.

- **Consequences of the Design Smell:** When using the relative path the loading and installation of the library can be done everywhere. But if we just put the name, this may impact the reuse of code or maintenance as the library cannot be accessed in the same way from everywhere if we do not specify the path. This design smell also impacts the reusability of the code, as the library could not be reused from anywhere without providing the full path.

- **Refactoring:** A good solution to remove this design smell would be to use relative path to load a library. **When using a native library, a relative path must be used to allow installation anywhere.** To avoid issues it is better to use a complete path as it points to the same location in a file system, regardless of the current working directory. This will ensure the reusability and improve the maintainability as in case of issues related to this library, any future developer can directly locate the library.

- **Benefits of the Refactoring:** One of the main benefits is to ensure the reusability as the library can be used from anywhere. Maintainers or developer can also easily locate the library. This also improves maintainability.

- **Examples:** We perceived examples of occurrences of this design smell when analyzing JNI systems. Only a few of the systems that we analyzed followed the practice of using a relative path. The systems `Conscrypt` and `JatoVm` are mainly relying on the relative path to load the library, while most of the systems that we analyzed only specify the name of the library. Figure 7.17 presents an example of refactoring to remove this design smell extracted from `JatoVm`.

```
/* Java */
public class JNITest extends TestCase {
 static {System.load("./test/functional/jni/libjnitest.so"); }
```

Figure 7.17 Refactoring - Not Using Relative Path to Load the Library

Memory Management Mismatch

- **Overview:** Data types differ between Java and C/C++. When using JNI, a mapping is performed between Java data types and data types used in the native code[15]. JNI handles Java objects, classes, and strings as reference types. JVM offers a set of predefined methods that could be used to access fields, methods, and convert types from Java to the native code. Those methods return pointers that will be used by the native code to perform the calculation. The same goes for reference types, the predefined methods used allow to either return a pointer to the actual elements at runtime or to allocate some memory and make a copy of that element. Thus, due to the differences of types between Java and C/C++, the memory will be allocated to perform respective type mapping between those programming languages. Memory leaks will occur if the developer forgets to take care of releasing such reference types.

- **Context:** We are implementing a multi-language system in which we are passing reference types from one language to another. Depending on the languages, these types may be considered as pointers when used in other languages.

- **Problem: The management of the types and memory is not the same from one language to another.** In some languages as C/C++, the management of the memory is not handled automatically. Depending on the language, it may be the developers' responsibility to care about the management of the memory. As in the case of JNI, if we are using a *String* we should be the one taking care of releasing it after its usage. However, developers do not have enough knowledge of the characteristics of programming languages involved. They are usually dealing with programming languages that automatically handle the management of the memory.

- **Bad Solution:** The bad solution would be to to not consider the differences and possible incompatibilities between different programming languages when managing the memory. Rely on the management of memory provided by a single programming

[15] https://www.developer.com/java/data/jni-data-type-mapping-to-cc.html

language without additional checks to confirm that they memory is correctly managed.

- **Consequences of the Design Smell:** The management of the memory may not be the same from one language to another. Memory leaks can occur if the developers forget to take care of releasing such reference types.

- **Refactoring:** To remove this design smell, a good solution would be to **always take care of the management of such references types.** It is better to assume that in foreign communication, the management of the memory is not always handled automatically, and may be considered by the developers. Especially the allocation and release of memory that needs to be explicitly done by the developers. It becomes their responsibility when dealing with more than one programming language.

- **Benefits of the Refactoring:** One of the main benefits is to avoid problems due to a leak of the memory. This also avoids performance issues and free the memory allocated to objects that are no longer used.

- **Examples:** Examples of occurrences of this design smell have been observed in few JNI systems and developers' documentation, where problems related to the leak of memory occurred due to not releasing the memory. Developers' should take care of such memory management in multi-language systems. For the JNI case, Java strings are handled by the JNI as reference types. Those reference types are not null-terminated C char arrays (C strings). When the Java string is converted to a C string, it simply becomes a pointer to a null-terminated character array. It is the developers' responsibility to explicitly release the arrays using the ReleaseString or ReleaseStringUTF functions. Figure 7.18 presents an example of the refactored solution to remove this design smell. As the example of occurrences of this design smell is the non-release of the memory using `ReleaseString` or `ReleaseStringUTF`.

```
/* C++ */
str = env->GetStringUTFChars(javaString, &isCopy);
if (0 != str) {env->ReleaseStringUTFChars(javaString, str);
str = 0; }
```

Figure 7.18 Refactoring - Memory Management Mismatch

Local References Abuse

- **Overview:** For any object returned by a JNI function, a local reference is created. JNI specification allows a maximum of 16 local references for each method. Developers should pay attention on the number of references created and always deleted the local references once not needed using `JNIDeleteLocalRef()`.

- **Context:** Depending on the programming language, the management of the memory is not the same. For this design smell, we are considering the references. For the Java code, JVM keeps an eye on the available references to the allocated memory regions. When JVM detects that an allocated memory region can no longer be reached by the application code. It releases the memory automatically through garbage collection leaving developer free from memory management. But JVM garbage collectors boundaries are limited to the Java space only.

- **Problem:** The lifespan of a local reference is limited to the native method itself. **Depending on the language, garbage collectors boundaries are limited to the specific space only,** so the garbage collector cannot free the memory that the application allocates in the native space. The management of the memory may differ from one language to another. Thus, developers should always consider taking care of memory when using local and global references. For the JNI case, memory models and their management defer between Java and C. It is the developers' responsibility to manage the application's memory in native space properly.

- **Bad Solution:** The bad solution would be to use local and global references without considering the management of the memory.

- **Consequences of the Design Smell:** If we do not consider the management of memory when using local references from one language to another, this can cause memory leaks.

- **Refactoring:** To remove this design smell, **always take care of releasing the memory once using global or local references and never assume that their release will be done automatically.** For the JNI case, it creates references for all object arguments passed into native methods, as well as all objects returned from JNI functions. These references will keep Java objects from being garbage collected. To make sure that Java objects can eventually be freed, the JNI by default creates local references. Local references become invalid when the execution returns from the native

method in which the local reference is created.

- **Benefits of the Refactoring:** One of the main benefits is to ensure releasing the memory once using global or local references. This avoids memory allocation bugs and makes sure to free the memory for reuse when no longer needed.

- **Examples:** Occurrences of this design smell have been observed in JNI systems and the good practice of releasing the memory has been discussed in several developers' documentation as well as the JNI specification. Each time we return an object by a JNI function, local references are created. For example, as shown in Fig. 7.19 calling GetObjectArrayElement() will return a local reference to each object in the array. It is important to delete each reference when it is no longer required. A native method must not store away a local reference and expect to reuse it in subsequent invocations. so whenever a state is to be maintained during JNI calls, global references is a must. However, JNI global references are prone to memory leaks, as they are not automatically garbage collected, and the programmer must explicitly free them but they are necessary. Depending on the programming language, to reuse a reference, the developer must explicitly create a global reference based on the local reference using the NewGlobalRef JNI API call. The global reference can be released when it is no longer needed using the *DeleteGlobalRef* function. An example of refactoring is: `env->DeleteLocalRef(globalObject)`.

```
/* C++ */
for (i=0; i < count; i++) {
jobject element = (*env)->GetObjectArrayElement(env, array, i);
if((*env)->ExceptionOccurred(env)) { break;}
```

Figure 7.19 Design smell - Local References Abuse

The figure 7.20 presents a pattern overview of the collected design smells and the relationships between them.

7.4 Threats to Validity

We now discuss threats to the validity of our methodology and the reported design smells.

Threats to internal validity: We accept this threat as we did not provide an exhaustive list of all existing design smells related to multi-language systems. However, we used well-

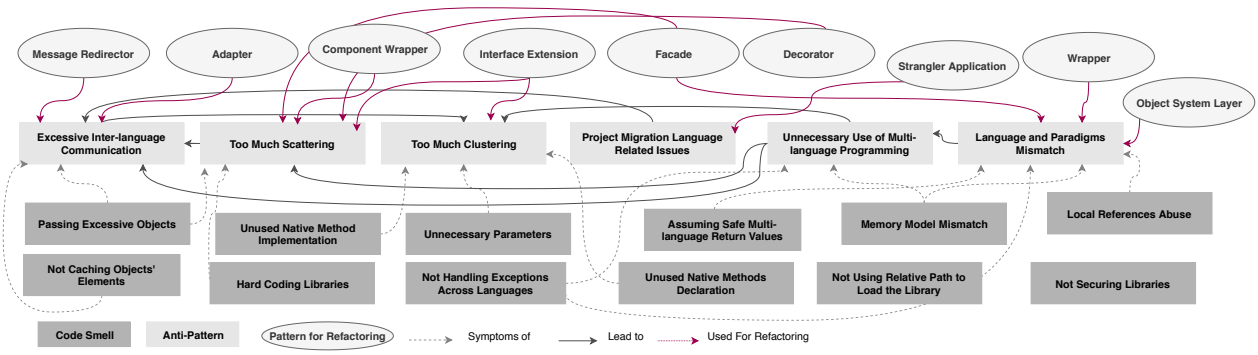


Figure 7.20 Pattern Overview Diagram - Relation Between Multi-language Design smells and Possible Design Patterns Applied for Refactoring

known, open-source repositories *GitHub* and *OpenHub* to identify and obtain multi-language systems. We also used well-used developers' documentations, bug reports, and developers' blogs, such as *StackOverflow*, *IBM Developers*, *developer.android*, and *Bugzilla* to extract practices and design smells. Hence, we limited threats to the internal validity, although we did not identify exhaustively all existing design smells. Moreover, we followed a systematic method to identify and report multi-language design smells. We published our catalogue in a pattern conference [34, 49]. The catalogue went through rounds of a shepherding process and Pattern Writers' Workshop in which we received meaningful comments and suggestions to refine and improve the design smells before being validated and published.

Threats to external validity: We observed each one of the design smells more than three times in multiple sources. However, depending on the languages, some of the design smells may not be existent or may have different consequences. Hence, we believe that our study is repeatable but could give different results for different programming languages.

Threats to reliability validity: We mitigate the threats to reliability by providing online access to all the data and scripts that we used to conduct this study. All the information are available in the companion website[16].

7.5 Chapter Summary

In this chapter, we have described the steps followed to collect, document, and define multi-language design smells. We reported about 15 design smells that we have identified and–or inferred from several resources *i.e.,* (literature, language specification, developers' blog, bug

[16] `https://github.com/ResearchML/Catalog-Patterns-MLS`

reports, open source projects). The design smells defined in this chapter could apply not only to JNI systems but also to other combinations of programming languages, to microservices, database, as to any other pieces of code in which such poor design or implementation choice could appear due to heterogeneity between components written in different programming languages (having diverse lexical, semantic, and syntactical programming rules). These design smells should help developers and researchers to handle the complexity of multi-language systems. We followed and adapted the template provided by Brown [27].

CHAPTER 8 A DETECTION APPROACH FOR MULTI-LANGUAGE DESIGN SMELLS

8.1 Chapter Overview

In the previous chapter, we defined a set of 15 multi-language design smells that we cataloged in order to help improve the quality of multi-language systems. Therefore, in this chapter, we propose an approach for the detection of the design smells introduced in Chapter 7. In this chapter, we start by presenting the implementation details of our approach. Then we present the rules used to define and detect the multi-language design smells. We also present the results of the evaluation of the proposed approach.

8.2 Approach Definition

Because no tools are available to detect design smells in multi-language systems, we build a new detection approach named `MLSInspect`. We used srcML[1], a parsing tool that converts source code into srcML, which is an XML format representation. The srcML representation of the source code adds syntactic information as XML elements into the source code text. Listing 8.2 presents the srcML representation of the code snippet presented in Listing 8.1. The main advantage of srcML, is that it supports different programming languages, and generates a single XML file combining source code files written in more than one programming language. Languages supported in the current version of srcML include Java, C, C++, and C#[2]. However, this could be extended to include other programming languages [125]. SrcML provides a wide variety of predefined functions that could be easily used through the XPath to implement specific tasks. XPath is frequently used to navigate through XML nodes, elements, and attributes. In our case, it is used to navigate through srcML elements generated as an XML representation of a given project. It allows access to all levels of information in a source-code document, *i.e.,* lexical, structural, syntactic and documentary. The ability to address source code using XPath has been applied to several applications [126].

Part of the content of this chapter is published in: Mouna Abidi, Md Saidur Rahman, Moses Openja, Foutse Khomh. "Are Multi-language Design Smells Fault-prone? An Empirical Study", *Published in Transactions on Software Engineering and Methodology (TOSEM), 2020, ACM.*

[1] `https://www.srcml.org/`

[2] `https://www.srcml.org/about.html`

Listing 8.1 Example of Java Code

```
public class HelloWorld {

    public static void main(String[] args) {
        // Prints "Hello World" to stdout
        System.out.println("Hello World");
    }
}
```

Listing 8.2 Example of Java Code Converted to SrcML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="0.9.5" language="Java"
    filename="HelloWorld.java"><class><specifier>public</specifier> class
    <name>HelloWorld</name> <block>{

    <function><specifier>public</specifier> <specifier>static</specifier>
        <type><name>void</name></type>
        <name>main</name><parameter_list>(<parameter><decl><type><name><name>String
     </name><index>[]</index></name></type>
         <name>args</name></decl></parameter>)</parameter_list> <block>{
        <comment type="line">// Prints "Hello World" to stdout</comment>
        <expr_stmt><expr><call><name><name>System</name><operator>.</operator>
        <name>out</name><operator>.</operator><name>println</name></name>
        <argument_list>(<argument><expr><literal type="string">"Hello
            World"</literal>
        </expr></argument>)</argument_list></call></expr>;</expr_stmt>
    }</block></function>

}</block></class></unit>
```

Our detection approach reports smell detection results for a given system in a CSV file. The report provides detailed information for each smells detected such as smell type, file location, class name, method name, parameters (if applicable). The approach also allows to post-process the results and create a summary file. The summary results provide a CSV file that details for each specific file or class the total number of occurrences of each type of smell, the date and other information related to that specific version of the system.

8.3 Detection Rules

The detection approach is based on a set of rules defined from the documentation of the design smells presented in Chapter 7. Those rules were validated by the pattern community during

the Writers' workshop organised in the context of the pattern conference *i.e.,* (EuroPlop) to document and validate the smells. For example, for the design smell *Local Reference Abuse*, we considered cases where more than 16 references are created but not deleted with the *DeleteLocalRef* function. The threshold 16 was extracted from developers blogs discussing best practices and the Java Native Interface specification [41][3, 4]. We present in the following the smell detection rules of the proposed approach. These rules are applied on the srcML elements generated as an XML representation of a given project. Since the smells described in this thesis are multi-language smells, the following rules detect the occurrences of smells by using the XPath queries in the srcML representation of the source code that contains Java and C/C++ native code.

1. **Rule 1: *Not Handling Exceptions***

$$(f(y) \mid f \in \{GetObjectClass, FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID\})$$
$$\textbf{\textit{AND}}\ (isExceptionChecked(f(y)) = False\ \textbf{\textit{OR}}\ ExceptionBlock(f(y)) = False)$$

   Our detection rule for the smell *Not Handling Exceptions* is based on the existence of call to specific JNI methods requiring an explicit management of the exception flow. The JNI methods (*e.g.,* FindClass) listed in the rule should have a control flow verification. The parameter y presents the Java object/class that is passed through a native call for a purpose of usage by the C/C++ side. Here, *isExceptionChecked* allows to verify that there is an error condition verification for those specific JNI methods, while *ExceptionBlock* checks if there is an exception block implemented. This could be implemented using Throw() or ThrowNew() or a return statement that exists in the method in case of errors.

2. **Rule 2: *Assuming Safe Return Value***

$$x := f(y) \mid f \in \{FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID\}\ \textbf{\textit{AND}}\ isErrrorChecked(x) = False\ \textbf{\textit{AND}}\ IsReturn(x) = True$$

   This rule is quite similar to the previous rule. However, it considers the return value from the native code. Indeed, the JNI methods called in this context are used for

[3] https://www.cnblogs.com/cbscan/articles/4733508.html

[4] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#global_local

specific calculation and the result then needs to be passed as a method return value to the Java side. Here, x presents the native variable used within the method to receive the returned value and perform computation on the Java side. *isErrrorChecked(x)* allows to verify if there is an error condition verification applied to the variable x that will be returned back to the Java code (*IsReturn(x)=True*). The use of the variable x as a return value by a native method without any check of its correctness will introduce smell of type *Assuming Safe Return Value* given other conditions hold.

3. **Rule 3: *Not Securing Libraries***

$$IsNative(Lib) = True \ \textbf{AND} \ loadedWithinAccessBlock\ (Lib) = False$$

This rule implies that in the Java code, a native library is used (*IsNative(Lib) = True*) and that this library is loaded outside a block *AccessController.doPrivileged* without a try and catch statements for safe handling of potential exceptions. This introduces smell of type *Not Securing Libraries*.

4. **Rule 4: *Hard Coding Libraries***

$$IsNative(Lib) = True \ \textbf{AND} \ AccessiblePath(Lib) = False \ \textbf{AND} \ OsBlock(m) = True$$

This rule implies that in the Java code, a native library (*Lib*) is used in a native method m and that the path used for accessing that library is an absolute path while the code loading the library depends on the operating systems. Here, the access to libraries is hard coded for specific operating system rather than implementing a platform independent access mechanism for libraries. This limits the portability of the code and may cause issues in accessing the libraries for different operating systems.

5. **Rule 5: *Not Using Relative Path***

$$IsNative(Lib) = True \ \textbf{AND} \ RelativePath(Lib) = False$$

This rule implies that in the Java code, a native library is used. However, the native library loaded from an absolute path and not from a relative path.

6. **Rule 6: *Too Much Clustering***

   NbNativeMethods (C) >= MaxMethodsThreshold ***AND*** *IsCalledOutside(m) = True*

   This rule detects cases where the total number of native methods (*NbNativeMethods*) within any class C is equal to or higher than a specific threshold while those methods m are used by other classes and not only the one where they are declared (*IsCalled-Outside(m) = True*). In our case, we used the default values for the threshold eight. However, all the thresholds could be easily adjusted.

7. **Rule 7: *Too Much Scattering***

   $$NBNativeClass(P) >= MaxClassThreshold$$
   $$\textbf{AND } (NbNativeMethods(C) < MaxMethodsThreshold \textbf{ AND } C \in P)$$

   The smell of type *Too Much Scattering* occurs when the total number of native classes in any package P (*NBNativeClass(P)*) is more than a specific threshold (*MaxClassThreshold*) for the number of maximum native classes. In addition, each of those native classes C contains a total number of native methods (*NbNativeMethods(C)*) less than a specific threshold (*MaxMethodsThreshold*) *i.e.,* the class does not contain any smell of type *Too Much Clustering*. We used default values for the threshold three for the minimum number of classes with each a maximum of three native method each.

8. **Rule 8: *Excessive Inter-language Communication***

   (NBNativeCalls(C,m) > MaxNbNativeCallsThreshold) ***OR***
   (NbNativeCalls(m(p)) > MaxNativeCallsParametersThreshold) ***OR***
   ((NBNativeCalls (m) > MaxNbNativeCallsMethodsThreshold) ***AND*** *IsCalledInLoop(m) = True)*

   The smell *Excessive Inter-language Communication* is detected based on the existence of at least one of these three scenarios. First, in any class C the total number of calls to a particular native method m exceeds the specified threshold (*NBNativeCalls(C,m) > MaxNbNativeCallsThreshold*). Second, the total number of calls to the native methods m with the same parameter p exceeds the specific threshold (*MaxNativeCallsParame-*

tersThreshold). Third, the total number of calls to a native method m within a loop is more than the defined threshold (*(MaxNbNativeCallsMethodsThreshold*).

9. **Rule 9: *Local References Abuse***

$$(NbLocalReference(f_1(y)) > MaxLocalReferenceThreshold) \textbf{ AND}$$
$$(f_1(y) \mid f_1 \in \{GetObjectArrayElement, GetObjectArrayElement, NewLocalRef,$$
$$AllocObject, NewObject, NewObjectA, NewObjectV, NewDirectByteBuffer,$$
$$ToReflectedMethod, ToReflectedField\}) \textbf{ AND}$$
$$(\nexists\ f_2(y) \mid f_2 \in \{DeleteLocalRef, EnsureLocalCapacity\})$$

The smell *Local References Abuse* is introduced when the total number of local references (*NbLocalReference(*f_1*(y))*) created inside a called method exceeds the defined threshold and without any call to method `DeleteLocalRef` to free the local references or a call to method `EnsureLocalCapacity` to inform the JVM that a larger number of local references is needed.

10. **Rule 10: *Memory Management Mismatch***

$$(mem \leftarrow f_1(y) \mid f_1 \in \{GetStringChars, GetStringUTFChars,$$
$$GetBooleanArrayElements, GetByteArrayElements,$$
$$GetCharArrayElements, GetShortArrayElements,$$
$$GetIntArrayElements, GetLongArrayElements,$$
$$GetFloatArrayElements, GetDoubleArrayElements, GetPrimitiveArrayCritical,$$
$$GetStringCritical\})$$
$$\textbf{AND}\ \ (\nexists\ f_2(mem) \mid f_2 \in \{ReleaseGetStringChars, ReleaseGetStringUTFChars,$$
$$ReleaseGetBooleanArrayElements, ReleaseGetByteArrayElements,$$
$$ReleaseGetCharArrayElements, ReleaseGetShortArrayElements,$$
$$ReleaseGetIntArrayElements, ReleaseGetLongArrayElements,$$
$$ReleaseGetFloatArrayElements, ReleaseGetDoubleArrayElements,$$
$$ReleaseGetPrimitiveArrayCritical, ReleaseGetStringCritical\})$$

As discussed earlier, JNI offers predefined methods to manage the access of reference types that are converted to pointers. These methods are used to create pointers and to

allocate the corresponding memory. The rule described here allows to detect the native implementation in which the memory was allocated by calling one of these allocation methods, however, the memory allocated was never released. The rule detects situations in which 'get' methods are used to allocate memory for specific JNI elements that are not released after usage by calling the corresponding 'release' methods.

11. **Rule 11: *Not Caching Objects***

$$
\begin{aligned}
&((Parameter(m,p) = Object)\ \textbf{AND}\\
&\quad((NbCalls(C,m) >= MaxNbCallsThreshold)\ \textbf{OR}\ (IsLoop(m) = True\\
&\qquad\textbf{AND}\ NoOfIterations >= MaxCountThreshold))\\
&\qquad\textbf{AND}\ (IsCalled(m, f_n(y)) = True)\\
&\quad\textbf{AND}\ (f_n(y) | f_n \in \{GetFieldID, GetMethodID, GetStaticMethodID\}))\\
&\textbf{OR}\ ((Parameter(m,p) = Object)\ \textbf{AND}\ (IsCalledInMethod(m, f_n) = True\\
&\qquad\textbf{AND}\ NbCalls(f_n(y)) >= MaxNbCallsThreshold)\ \textbf{AND}\\
&\qquad(f_n(y) | f_n \in \{GetFieldID, GetMethodID, GetStaticMethodID\}))
\end{aligned}
$$

This rule allows to detect occurrences of the smell *Not Caching Objects* based on two situations. The first one is where the total number in which ids related to the same object p are looked up for the same class C through JNI allocation methods is greater than or equal to a specific threshold or the method is called within a loop. Indeed, the ids returned for a given class C remain the same for the lifetime of the JVM execution. Considering that we have a native method m and one of its parameter p is a Java object (*Parameter(m,p)=Object*), this type is considered in the native code as a reference type. Thus, unlike primitive types, its element could not be accessed directly by the native code but should be accessed through the usage of the methods defined in (*IsCalled(m,*$f_n(y)$) = True). In this first scenario, the total number of calls from the Java code to a native method m that is defined in a class C exceeds a specific threshold (*i.e., NbCalls(C,m) >= MaxNbCallsThreshold*) or the method is called within a loop. In the second scenario, the number of times the same id for an object p is looked up inside the same method m (*IsCalledInMethod(m,* f_n*)=True*) more than a given threshold even if the method m is called only once (*NbCalls(*$f_n(y)$*) >= MaxNbCallsThreshold*). This last scenario includes the total number of calls to the predefined methods (*NbCalls(*$f_n(y)$) independent of the total number of calls to the method itself.

12. **Rule 12: *Excessive Objects***

$$(Parameter(m,p) = Object) \textbf{ AND } (IsCalledInMethod(m, f_1) = True) \textbf{ AND}$$
$$(NbCalls(f_1(y)) >= MaxNbCallsThreshold) \textbf{ AND}$$
$$(f_1(y) | f_1 \epsilon \{GetObjectField, GetBooleanField, GetByteField, GetCharField,$$
$$GetShortField, GetIntField, GetLongField, GetFloatField, GetStaticObjectField\})$$
$$\textbf{AND } (\nexists\, f_2(y) \mid f_2 \in \{SetObjectField, SetBooleanField, SetByteField, SetCharField,$$
$$SetShortField, SetIntField, SetLongField, SetFloatField, SetStaticObjectField\})$$

This rule identifies situations in which a JNI object is passed as a parameter (*Parameter(m,p)=Object*) to the native code. In this context the total number of calls to allocation methods to retrieve its field id in the same method is higher than a specific threshold (*i.e., NbCalls($f_1(y)$) >= MaxNbCallsThreshold*), without a call to corresponding set functions to set the object fields by the native code. However, as described in Chapter 7, having the total number of calls to allocation methods higher than the threshold is not considered as a smell only in situations where the purpose of those calls was to set the object fields by the native code.

13. **Rule 13: *Unused Method Implementation***

IsNative(m) = True ***AND*** *IsDeclared(m) = True* ***AND*** *IsImplemented(m) = True* ***AND*** *IsCalled(m)=False*

This rule allows to capture the native functions m (*IsNative(m) = True*) implemented in the C/C++ (*IsImplemented(m) = True*), declared in Java with the keyword *native* but never used in the Java code (*IsCalled(m)=False*). It looks for the native methods that are declared using the keyword *native* with a header in the Java code and looks for the corresponding native implementation nomenclature.

14. **Rule 14: *Unused Method Declaration*:**

IsNative(m)=True ***AND*** *IsDeclared(m)=True* ***AND*** *IsImplemented(m)=False*

Native functions declared in Java with the keyword *native* (*IsDeclared(m) = True*) that are not implemented in C/C++ (*IsImplemented(m)=False*). This rule allows to retrieve the native methods that are declared with a header in the Java code using the keyword *native* and checks for the corresponding implementation nomenclature. However, those methods were never used or even implemented in the C/C++ code.

15. **Rule 15: *Unused Parameters***

$$(IsNative(m(p)) = True \ \textbf{\textit{AND}} \ IsDeclared(m(p)) = True$$
$$\textbf{\textit{AND}} \ IsImplemented(m(p)) = True \ \textbf{\textit{AND}} \ IsParameterUsed(p) = False$$

This rule reports the method parameters that are used in the Java native method declaration header using the keyword *native* (*IsDeclared(m(p))=True*). However the parameter is never used in the body of the implementation of the methods, apart from the first two arguments of JNI functions in C/C++. The rule checks if the parameter p is used in the corresponding native implementation (*IsParameterUsed(p) = False*).

8.4 Evaluation of the Detection Approach

In order to evaluate the detection approach, we started by selecting the object systems that will be used for the evaluation. We considered six open source projects that we analyzed in a prior study and that were part of the definition of the design smells as presented in Chapter 7 [34, 49, 120]. Those systems are open source projects hosted in GitHub, they are highly active, and have the characteristic of being developed with Java and C/C++. Another selection criteria was that those systems have different size and belong to different domains. Table 8.1 presents the studied systems. Similar to previous studies evaluating detection approaches for design patterns and design smells [78, 90, 94, 95, 127], we decided to measure the recall and precision of the detection approach as they they are the adequate performance metrics for such evaluation. To assess the recall and precision of our detection approach, we evaluated the results of the approach at the first level by creating dedicated unit tests for the detector of each type of smell to confirm that the approach is detecting the smells introduced in our pilot project. The pilot project was the project we developed with the occurrences of the smells along with the clean code without any smell to test and validate our approach. This explains the 100% precision and 100% recall for all the smells. We relied on six open source projects used in previous works [34, 49] on multi-language design smells. For each of the systems, we manually identified occurrences of the studied design smells. Two of the research team members independently identified occurrences of the design smells in JNI open source projects, and resolved disagreements through discussions with the whole research team. Using the ground truth based on the definition of the smell and the detection results, we computed *precision* and *recall* as presented in Table 8.1 to evaluate our smell detection approach. Precision computes the number of true smells contained in the results

of the detection tool, while recall computes the fraction of true smells that are successfully retrieved by the tool. From the six studied systems, we obtained a precision between 88% and 99%, and a recall between 74% and 90%. We calculate precision and recall based on the following Equations (8.1) and (8.2) respectively:

$$Precision = \frac{\{existing\ true\ smells\} \bigcap \{detected\ smells\}}{\{detected\ smells\}} \tag{8.1}$$

$$Recall = \frac{\{existing\ true\ smells\} \bigcap \{detected\ smells\}}{\{existing\ true\ smells\}} \tag{8.2}$$

We present in Table 8.2 the results of the evaluation of the performance of our design smell detection approach. As explained earlier, for the pilot project, we have 100% precision and 100% recall for all the smells. For other projects, the precision was evaluated by a manual inspection of all the identified occurrences of multi-language design smells by the approach. The recall was evaluated by a manual investigation of the Java and native code (C/C++) to capture occurrences may have not been detected by the approach. The reasons for false positives (FP) and false negatives (FN) are mainly related to the alternative implementations of the multi-language code that do not follow JNI specification guidelines and therefore are not currently covered by our approach. Indeed, our approach considers the JNI implementation with the appropriate naming convention as described in the JNI specification (*e.g.,* using the *native* keyword in the Java native method declaration, using JNIENV, JNIEXPORT, JNICALL, and Java_ClassName_methodname) [128]. Our approach may present some limitations for the smell *Local References Abuse* in situations in which some specific methods are used to ensure the memory capacity. However, as per our manual analysis when defining the smells, those methods are not considered relevant to detect the smell and are not frequently used. We are aware that in similar situation, the approach may result in false positives. For the smells *Assuming Safe Return Value* and *Not Handling Exceptions*, the false negatives in *Conscrypt* were related to an intermediate step that made the detection harder. In this step, the native value was checked before returning it to the Java code.

Table 8.1 Validation of the Smell Detection Approach

Systems	True Positive	False Positive	False Negative	Recall	Precision
openj9	3293	137	250	93%	96%
rocksdb	922	50	136	87%	95%
conscrypt	556	29	133	80%	95%
pilot project	32	0	0	100%	100%
pljava	511	5	53	90%	99%
jna	375	50	127	74%	88%
jmonkey	2210	142	185	92%	94%

Table 8.2 Validation Results for Each Type of Smells

Project	**Evaluation**	EIC	TMC	TMS	UMD	UMI	UP	ASRV	EO	NHE	NCO	NSL	HCL	NURP	MMM	LRA
PilotProject	FP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Precision	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	FN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
conscrypt	FP	0	0	-	6	-	15	3	-	5	-	0	-	0	0	0
	Precision	100%	100%	-	98%	-	95%	0%	-	0%	-	100%	-	100%	100%	100%
	FN	0	0	-	0	-	132	0	-	0	-	0	-	0	0	1
	Recall	100%	100%	-	100%	-	67%	100%	-	100%	-	100%	-	100%	100%	80%
pljava	FP	4	0	0	1	0	0	-	-	0	-	-	-	-	0	-
	Precision	95%	100%	100%	98%	100%	100%	-	-	100%	-	-	-	-	100%	-
	FN	3	0	0	42	0	1	-	-	0	-	-	-	-	8	-
	Recall	96%	100%	100%	67%	100%	99%	-	-	100%	-	-	-	-	50%	-
openj9	FP	9	0	0	29	-	95	0	-	3	-	0	-	0	1	0
	Precision	96%	100%	100%	95%	-	95%	100%	-	98%	-	100%	-	100%	94%	100%
	FN	37	0	0	40	-	76	4	-	86	-	0	-	0	4	1
	Recall	85%	100%	100%	94%	-	96%	66%	-	63%	-	100%	-	100%	81%	80%
rocksdb	FP	24	2	5	2	-	17	0	-	0	-	0	0	0	0	-
	Precision	96%	96%	92%	88%	-	93%	100%	-	100%	-	100%	100%	100%	100%	-
	FN	91	4	5	0	-	31	0	-	0	-	3	0	0	2	-
	Recall	86%	92%	92%	100%	-	88%	100%	-	100%	-	73%	100%	100%	75%	-
jmonkey	FP	8	0	0	12	-	59	32	0	31	-	0	-	0	-	-
	Precision	95%	100%	100%	95%	-	96%	88%	100%	89%	-	100%	-	100%	-	-
	FN	75	0	0	86	-	13	7	0	0	-	0	-	0	-	-
	Recall	65%	100%	100%	72%	-	99%	97%	100%	100%	-	100%	-	100%	-	-
jna	FP	5	0	0	-	-	43	-	-	0	-	0	0	-	-	2
	Precision	84%	100%	100%	-	-	88%	-	-	100%	-	100%	100%	-	-	78%
	FN	22	0	0	-	-	64	-	-	1	-	2	0	-	-	2
	Recall	54%	100%	100%	-	-	83%	-	-	83%	-	71%	100%	-	-	78%
Average	Precision	94%	99%	98%	94%	-	94%	72%	-	81%	-	100%	-	100%	98%	92%
	Recall	81%	98%	98%	86%	-	88%	90%	-	91%	-	88%	-	100%	76%	79%

8.5 Discussion and Threats to Validity

Our proposed detection approach presents some limitations. The recall and precision vary depending on the type of design smells and mainly based on the naming convention used to implement the JNI projects. For the smells *Unused Parameters* and *Unused Method Declaration*, when evaluating the recall and precision, we noticed that our approach was not always able to correctly match the java and corresponding native implementation. This was mainly due the syntax used in the C implementation that is not completely following the JNI specification for the naming convention (*e.g., Pljava_jobject pljava_DualState_key*). For *Local References Abuse*, we are not considering situations in which predefined methods could be used to limit the impact of this design smell, *i.e.,* PushLocalFrame[5], and PopLocalFrame[6]. These methods were excluded because during a manual validation when defining the smells, we found that those methods do not always prevent occurrences of the design smells and the inclusion of those methods may result in false negatives. Our detection approach also presents some limitations in the detection of *Not Using Relative Path*, particularly in situations where the path could be retrieved from a variable or concatenation of strings. However, this was not captured as a common practice in the analyzed systems. The same goes for the smell *Memory Management Mismatch*. Indeed, we implemented a simple detection approach that could be applied to detect the smells following the definitions and rules presented in this Chapter. Thus, this could not be generalized to all memory allocation issues. The detection approach relies on rules specific to the JNI systems. Thus, other native methods that could be implemented without considering JNI guidelines could lead to false positives and false negatives. To reduce threats to the validity of our work, we manually verified instances of smells reported by our detection approach on six open source projects along with our pilot project and measured the recall and precision of our detection approach.

8.6 Chapter Summary

Building on the catalogue provided in the previous chapter, we presented in this chapter an approach that is able to detect occurrences of multi-language design smells in the context of JNI systems. The approach was evaluated with six JNI systems and results show that it has a minimum precision and recall of 88% and 74%, respectively.

[5] `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#PushLocalFrame`

[6] `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#PopLocalFrame`

CHAPTER 9 PREVALENCE AND IMPACT OF MULTI-LANGUAGE DESIGN SMELLS ON FAULT-PRONENESS

9.1 Chapter Overview

In this chapter we study the prevalence of multi-language design smells presented in Chapter 7 and their impacts on software fault-proneness. Despite the importance and increasing popularity of multi-language systems, there is no existing study that empirically evaluates the impacts of design smells on software fault-proneness. Through this thesis, we aim to fill this gap in the literature. Based on our approach presented in Chapter 8, we detect occurrences of 15 multi-language design smells in 98 releases of nine open source multi-language projects (*i.e., VLC-android*, *Conscrypt*, *Rocksdb*, *Realm*, *Java-smt*, *Pljava*, *Javacpp*, *Zstd-jni*, and *Jpype*). We focus on the analysis of JNI systems because they are commonly used by developers and also introduce several challenges [10–12]. Our results from Chapter 5 also report that JNI systems are widely studied in the literature due to their challenges.

9.2 Study Design

In this section, we present the methodology we followed to conduct this study. Figure 9.1 provides an overview of our methodology.

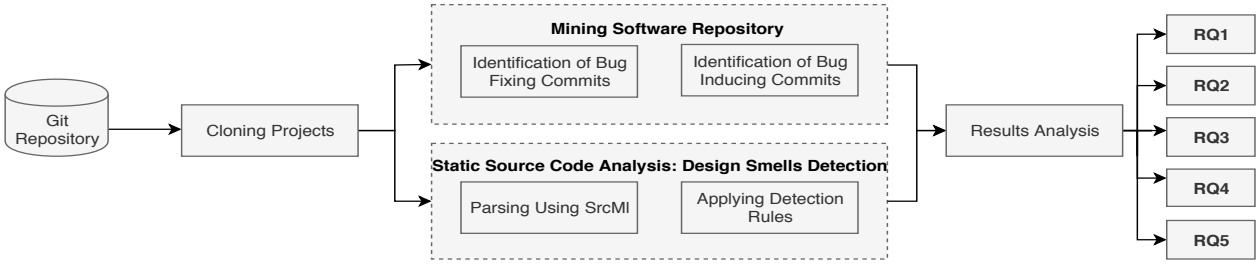


Figure 9.1 Schematic Diagram of the Study

Part of the content of this chapter is published in: Mouna Abidi, Md Saidur Rahman, Moses Openja, Foutse Khomh. "Are Multi-language Design Smells Fault-prone? An Empirical Study", *Published in Transactions on Software Engineering and Methodology (TOSEM), 2020, ACM.*

9.2.1 Setting Objectives of the Study

We started by setting the objective of our study. Our objective is to investigate the prevalence of multi-language design smells in the context of JNI systems and the relation between those smells and software fault-proneness. We also aim to investigate what kind of activities once performed in smelly files are more likely to introduce bugs. The quality focus in this study is the occurrence of bugs due to the presence of design smells in JNI systems. The perspective is that of researchers, interested in the quality of JNI systems, and who want to get evidence on the impact of design smells on the software fault-proneness. Also, these results can be of interest to professional developers performing maintenance and evolution activities on JNI projects and who need to take into account and forecast their effort, since like for mono-language projects, the presence of fault-prone files is likely to increase the maintenance effort and cost. These results are also of interest to testers since they need to know which files are more important to test. Finally, they can be of interest to quality assurance teams or managers who could use design smells detection techniques to assess the fault-proneness of in-house or to-be-acquired source code, to better quantify the cost-of-ownership of JNI systems.

We defined our research questions as follows:

RQ9.1: **Do Multi-language design smells occur frequently in open source projects?**
Several articles in the literature discussed the prevalence, detection, and evolution of design smells in the context of mono-language systems [31, 127]. Occurrences of design smells may hinder the evolution of a system by making it hard for developers to maintain the system. The detection of smells can substantially reduce the cost of maintenance and development activities. However, most of those research are focusing on mono-language systems. Thus, we decided to fill this gap in the literature and investigate the frequency of design smells in the context of multi-language systems. This research question is preliminary to the remaining questions. It aims to examine the frequency and distribution of multi-language design smells in the studied projects and their evolution over the releases of the project. We defined the following null hypothesis: *H_1: there are no occurrences of the multi-language design smells presented in Chapter 7 in the studied projects.*

RQ9.2: **Are some specific multi-language design smells more frequent than others in open source projects?**
Given that multi-language design smells are prevalent in the studied systems, it is im-

portant to know the distribution and evolution of the different types of smells for a better understanding of the implication of their presence for maintenance activities. Developers are likely to benefit from knowing the dominating smells to treat them in priority and avoid introducing such occurrences. Consequently, in this research question, we aim to study whether some specific types of design smells are more prevalent than others. We are also interested in the evolution of each type of smells over the releases of the project. We aim to test the following null hypothesis: *H_2: The proportion of files containing a specific type of design smell does not significantly differ from the proportion of files containing other kinds of design smells.*

RQ9.3: **Are files with multi-language design smells more fault-prone than files without?**

Prior works reported that classes containing design smells in mono-language systems are more prone to faults than other classes [23, 24]. Due to components written in different languages, multi-language systems may have more complexities in architecture and inter-component interactions. Given the known impacts of design smells on mono-language systems, it is thus important to investigate the impacts of multi-language design smells on the corresponding software systems. To examine this, we aim to investigate whether source files containing multi-language design smells are more likely to experience faults than files without smells. We investigate whether files with multi-language design smells are more fault-prone than others by testing the null hypothesis: *H_3: The proportion of files experiencing at least one bug does not significantly differ between files with design smells and files without.*

RQ9.4: **Are some specific multi-language design smells more fault-prone than others?**

During maintenance and quality assurance activities, developers are interested in identifying parts of the code that should be tested and–or refactored in priority. Hence, we are interested in identifying design smells that are more fault-prone than others. Thus, we defined the null hypothesis. *H_4: There is no significant difference between the impacts of different kinds of multi-language design smells on the fault-proneness of files containing those smells.*

RQ9.5: **What are the activities that are more likely to introduce bugs in smelly files?**

During the maintenance of a project, having knowledge of possible risky activities could help developers and managers to reduce the risk of bugs. They could benefit from that knowledge to capture activities that should be performed with caution in smelly files. Hence, we are interested in identifying what kinds of activities once performed in smelly files are likely to introduce bugs. Capturing such information could provide insights about what kind of activities could increase the risk of bugs in smelly files.

9.2.2 Data Collection

In order to address our research questions, we selected nine open source projects hosted on GitHub. We decided to analyze those nine systems because they are well maintained, and highly active. Another criteria for the selection was that those systems have different size and belong to different domains. They also have the characteristic of being developed with more than one programming language. While those systems contain different combinations of programming languages, for this study, we are analyzing the occurrences of design smells for only Java and C/C++ code. For each of the nine selected subject systems, we selected a minimum of 10 releases. For projects with relatively frequent releases and comparatively a small volume of changes per release, we extended our analysis to a few extra releases to cover a longer evolution period for our analysis. Tables 9.2 and 9.3 summarise the characteristics of the subject systems and releases. We also provide the percentage of the Java and C/C++ code in the studied projects in Table 9.2.

9.2.3 Data Extraction

To answer our research questions, we first have to mine the repositories of the nine selected systems to extract information about the occurrences of smells existing in each file and also the bugs reported for those systems.

Table 9.1 Research Objectives and Research Questions

Research Objectives	Methodology
Objective 1: Investigate the prevalence of multi-language design smells	RQ9.1 and RQ9.2
Objective 2: Study the relationship between multi-language design smells and fault-proneness	RQ9.3 and RQ9.4
Objective 3: Identifying fault-inducing activities	RQ9.5

Table 9.2 Overview of the Studied Systems

Systems	Domain	#Releases	#Commits	#Issues	LOC	Java	C/C++
Rocksdb[1]	Facebook Database	189	8375	1748	487853	11%	83.1%
VLC-android[2]	Media Player and Database	176	12697	1091	125037	10.1%	6.7%
Realm[3]	Mobile Database	169	8244	3886	171705	82%	8.1%
Conscrypt[4]	Cryptography (Google)	32	3874	186	91765	85.3%	14%
Pljava[5]	Database	27	1236	123	71910	67%	29.7%
Javacpp[6]	Compiler	34	658	269	28713	98%	0.6%
Zstd-jni[7]	Data Compression (Facebook)	36	423	78	72824	4.3%	92.1%
Jpype[8]	Cross Language Bridge	14	895	305	53826	7.8%	58%
Java-smt[9]	Computation	22	1822	146	42049	88%	4.6%

Table 9.3 Analyzed Releases in Each Project

Systems	#Releases Analyzed	Releases	Analysis Periods
Rocksdb	10	5.0.2 - latest release	2017-18-01 - 2019-14-08
VLC-android	10	3.0.0 - latest release	2018-08-02 - 2019-13-09
Realm	10	0.90.0 - 5.15.0	2016-03-05 - 2019-04-09
Conscrypt	11	1.0.0.RC11 - 2.3.0	2017-25-09 - 2019-25-09
Pljava	12	1_2_0 - latest release	2015-20-11 - 2019-19-03
Javacpp	13	0.5 - 1.5.1-1	2013-07-04 - 2019-05-09
Zstd-jni	11	0.4.4 - latest release	2015-17-12 - 2019-19-08
Jpype	11	0.5.4.5 - latest release	2013-25-08 - 2019-13-09
Java-smt	10	0.1 - 3.0.0	2015-27-11 - 2019-30-08

9.2.4 Detection of Design Smells

Detection Approach: We used the multi-language smell detection approach proposed in Chapter 8. The detection approach supports 15 types of multi-language smells as described in the recently published catalogue of multi-language design smells and detailed in Chapter 7 [34, 49]. The detection approach is based on a set of predefined detection rules extracted from the definition and documentation of the design smells [129].

9.2.5 Detection of Fault-inducing Commits

The studied systems use Github as the issue tracker. We used Github APIs and PyDriller to mine the software repositories and get the list of all the commit logs and resolved issues for the systems [130]. PyDriller provides a set of APIs to extract information from Git repositories. These include important historical information regarding commits, developers, and modifications. PyDriller is very convenient for mining software repositories to analyze changes or bugs. It relies on the SZZ algorithm [131] to detect changes that introduce faults. We used PyDriller because this approach was not only evaluated regarding existing tools but also with experiments involving developers [130]. We started by retrieving all the information related to the projects. We analyzed all commit messages to identify the fault-fixing commits. We used a set of error related keywords to identify commits related

to fault-fixing using a heuristic similar to that presented in the study by Mockus and Votta [132]. Our list of keywords includes “fix”, “crash”, “resolves”, “regression”, “fall back”, “assertion”, “coverity”, “reproducible”, “stack-wanted”, “steps-wanted”, “testcase”, “fail”, “npe”, “except”, “broken”, “bug”, “differential testing”, “error”, “addresssanitizer”, “hang”, “permaorange”, “random orange”, “intermittent”, “steps to reproduce”, “assertion”, “leak”, “stack trace”, “heap overflow”, “freez”, “str:”, “problem ”, “overflow”, “avoid“, “ issue”, “workaround”, “break”, and “stop”. To retrieve fault-inducing commits, given a commit, PyDriller returns the set of commits that previously modified the lines from the files included in the given commit. It applies the SZZ algorithm to find the commit when the bug was initially introduced as used in some earlier studies [110,133,134]. To locate the fault-inducing commits, PyDriller algorithm works as follows: for every file in the commit, it obtains the diff between the files, then obtains the list of all deleted lines. It then blames the file to obtain the commits where the deleted lines were changed. We tagged fault-inducing commits as *buggy*. We used this tag later to distinguish between files containing bugs and files without.

Since Pydriller’s SZZ implementation was not previously evaluated, we manually examined the bug inducing commits retrieved by Pydriller from two of our studied projects, *Pljava* and *Zstd-jni*. We performed this manual analysis in two steps. First, we executed an existing implementation of the SZZ algorithm available on GithHub[10] on *Pljava* and *Zstd-jni*. We compared its reported results with the results obtained from Pydriller. For each bug fixing commit, we manually verified if the related bug inducing commit reported by Pydriller matches with the one reported by SZZ. For that, we used two labels (True or False) to distinguish between the bug inducing commits that match with those retrieved by SZZ and those that do not match. Next, we manually verified if the changes in the bug inducing commits reported by Pydriller were indeed related to the changes performed in the corresponding bug fixing commits. We also analyzed the commit messages. We labeled each of the bug inducing commits with three tags (True, False, and Unclear). We used the tag True in situations in which we were convinced that the change performed in the bug fixing was indeed related to the changes applied in the bug inducing. We assigned False in situations in which it was evident that the changes are not related, and Unclear, in situations in which it was not completely evident to assign a True or False tag. We analyzed for *Pljava* and *Zstd-jni* respectively a total of 113 and 96 bug-fixing commits. We performed a cleaning process on those commits and removed the commits related to typos fixing and merge commits. We kept in our validation bug-fixing commits with their corresponding bug-inducing commits. Our final dataset results on 61 bug-fixing commits for *Pljava* and 66 bug-fixing commits for *Zstd-jni*. From our manual validation of fault-inducing commits reported by Pydriller for

[10]https://github.com/saheel1115/szz

Pljava and *Zstd-jni*, we found respectively precision values of 78.94% and 70.83%. Those values are computed considering only the True and False tags for Java and C/C++ files resulting from our manual validation. From the comparison between Pydriller'szz and the recent implementation of szz, we found for *Pljava* and *Zstd-jni*, respectively precision values of 85% and 80%. We did not include in our validation the recall because in our study we are considering only JNI code. However, szz is considering the whole project in general without considering multi-language interactions. So, those results may not generalize to the whole system. However, the results of our manual validation do not directly contribute to any of our empirical findings, and we did this validation as a complementary step to reduce the threats to validity of our study. We also analyzed changes related to multi-language programming. Indeed, in many situations, the Java and C/C++ code are changed within the same commit. This was helpful to validate the bug inducing commits involving Java and C/C++ code.

9.2.6 Analysis Method

We present in the following the analysis performed to answer our research questions.

9.2.7 Analyzing the Prevalence of Design Smells

We investigate the presence of 15 different kinds of design smells. Each variable $\mathrm{s}_{i,j,k}$ reflects the number of times a file i has a smell j in a specific release r_k.

For RQ9.1, since we are interested to investigate the prevalence of multi-language design smells, we aggregate these variables into a Boolean variable $\mathrm{s}_{i,k}$ to indicate whether a file i has at least any kind of smells in release r_k. We calculate the percentage of files affected by at least one of the studied design smells, s_j. We use our detection approach to detect occurrences of multi-language design smells following the methodology described earlier. For each file, we compute the value of a variable $Smelly_{i,r}$ which reflects if the file i has a least one type of smell in a specific release r. This variable takes 1 if the file contains at least one design smell in a specific release r, and 0 otherwise. Similarly, we also compute the value of variable $Native_{i,r}$ which takes 1 if the file i of a specific release r is native and 0 if not. Since our tool is focusing on the combination of Java and C/C++, we compute for each release the percentage of files participating in at least one design smells out of the total number of JNI files (files involved in Java and C/C++).

For RQ9.2, we investigate whether a specific type of design smells is more prevalent in the studied systems than other types of design smells. For that, we calculate for each system the percentage of files affected by each type of the studied smells j. For each file i and for

each release r, we defined a flag $Smelly_{i,j,r}$ which takes the value 1 if the release r of the file i contains the design smell type j and 0 if it does not contain that specific smell. Based on this flag, we compute for each release the number of files participating in that specific smell. We also calculate the percentage of smelly files containing each type of smell. Note that the same file may contain more than one smell. We investigate the presence of 15 different kinds of smells. We also compute the metric $s_{i,j,k}$ which reflects the number of occurrences of smells of type j in a file i in a specific release r_k.

9.2.8 Analyzing the Impacts of Smells on Bugs

For RQ9.3, we focus on each of the smells to study whether the proportion of files containing at least one bug, significantly differs between files containing smells and files without smells. We consider the number of bugs $c_{i,k}$ a file i encountered between releases r_k and r_{k+1}, and convert $c_{i,k}$ into a Boolean variable $f_{i,k}$ (true if the file underwent at least one bug, false otherwise). We rely on Fisher's exact test [135] to check whether the proportion of buggy files varies between two samples (files with and without smells). This test is useful for categorical data that result from the classification of objects. We also calculate the *odds ratio* (OR) indicating the likelihood for an event (bug in our case) to occur.

We use the *fisher_exact* function of the *stats* module from *scipy* Python package to compute the odds ratio and the p-value for statistical significance of the test. By processing the commits and bug information we set different flags for each of the source files. As mentioned earlier, the *smelly* flag takes the value 1 if the associated source file contains at least one design smell of any type, and 0 otherwise. The flag *buggy*, takes the value 1 if the associated source file was identified by SZZ algorithm as related to a fault-inducing commit, and 0 otherwise. Now, for a given release of a system, we consider all JNI source files for analysis. We count the number of buggy and non-buggy files with design smells. Similarly, we also count the number of buggy and non-buggy files without design smells. With these four values, we form the 2x2 contingency table for Fisher's exact test.

For RQ9.4, we investigate the relationship between different types of design smells with fault-proneness. Unlike using logistic regression for prediction purposes ([31, 136]), we use it to examine whether some types of design smells are more related to fault-proneness. Our analysis approach is similar to the one presented by Khomh *et al.* [31] where they investigate the impacts of different types of anti-patterns on change- and fault-proneness using logistic regression model.

In our regression model, independent variables are the number of occurrences of each type of design smells. The dependent variable is the flag (*buggy*) representing the presence or

absence of bugs. Thus, the dependent variable is dichotomous and assumes values either 0 (non-buggy) or 1 (buggy). For each system, we build a regression model and analyze the model coefficients and p-values for individual types of smells. Each row in our data set contains the values of the metrics (number of occurrences) for different smells, file size (LOC), number of previous bug-fix, code churn, and the bug status (1 or 0).

Each logistic regression model gives the log odds (regression coefficient estimate) of individual independent variables and their corresponding p-values for a particular system. We rank the smells based on the model coefficients and the corresponding p-values. We select files that contain at least one smell of any type. For a given type of smell, if the model coefficients show higher log odds (LO) of bugs in the majority (in percentage) of the systems, we consider the smell to be related to fault-proneness. It is important to mention that we analyzed the data for correlation among smells and dropped one independent variable from each pair of highly correlated variables. This ensures a non-redundant set of variables for the logistic regression models. From a highly correlated pair, we keep the variable representing smell type with a comparatively higher overall prevalence in the studied systems. Because the following metrics are known to be related to fault-proneness [24, 137, 138], we add the file size, code churn, and the number of the previous occurrence of faults to our model, to control their effect. Here, (i) LOC: number of lines of code in the file at that specific release; (ii) Code Churn: the sum of lines added and removed in the file before that specific release; (iii) No. of Previous-Bugs: the number of faults fixing related to that file before the particular release r.

9.2.9 Topic Modeling to Identify Fault-inducing Activities

For **RQ9.5**, we are interested to investigate what kind of activities once performed in smelly files, are more likely to introduce bugs than other activities. We decided to analyze the commit messages that developers described when they performed a change that was captured by the SZZ algorithm as a fault-inducing commit. Having knowledge about those activities, developers could pay more attention to avoid introducing additional bugs. We collect all the fault-inducing commits messages related to smelly files as described earlier. We then classify those commit messages into different topics of activities based on the keywords mentioned by developers using a mix of automated and manual techniques. We decided to apply both topic modeling strategies and manual text analysis. Similar to previous work [17, 139], we used Latent Dirichlet Allocation (LDA) [140], a well-known topic modeling algorithm to analyze the text and extract a set of frequently co-occurring words (*i.e.,* topics).

To generate the topic of activities introducing bugs, we combine both manual and automated approaches to build a categorization of risky activities. Based on developers' commit

messages, similar to previous work [141], we used *MALLET*[11], a specific type of LDA implementation to generate a set of topics based on frequently co-occurring words. We removed stop words using MALLET stop words list (*e.g.,* a, the, is, this, punctuation marks, numbers, and non-alphabetical characters). We also used Porter stemmer to reduce words to their root words (*e.g.,* programmer became program) [142]. Since our objective is to study the activities that could introduce bugs once performed in smelly files, we limited our study to the smelly files in which a bug was introduced (Flag =1). Thus, our data-set resulted in 2707 commit messages. We manually inspected the commit messages to estimate the number of possible topics for each system and also to assign a meaningful name to each topic. Once the number of possible topics was fixed, we used a python script that takes as input the list of all the commits messages in a CSV file and returns the list of commit messages with common keywords that could be used to build the topic. Two of the research team went through all the topics extracted for all the systems, and manually assigned meaningful names to each topic. The name of the topic was decided based on manual inspections of the commit messages and the keywords used to build that topic. We relied on the keywords generated by MALLET but also on frequent keywords captured during the manual analysis. We manually analyzed a total of 500 commits. To resolve the disagreements, two of the research team went through those commit messages and discussed the main topics of activities performed on those commits. Through those analysis, we aim to capture the possible types of activities that were described in the commit messages of the bug-inducing commits.

9.2.10 Study Results

In this section, we report on the results of our study by addressing the five research questions defined in Section 9.2. We focus on the three key research objectives of our study. First, research questions RQ9.1 and RQ9.2 investigate the prevalence of the multi-language design smells in software systems. Then, research questions RQ9.3 and RQ9.4 evaluate the impacts of the design smells on the fault-proneness of JNI systems. Finally, RQ9.5 investigates fault-inducing activities. We present additional insights into the findings from the research questions later in Section 9.2.11.

RQ9.1: Do Multi-language design smells occur frequently in open source projects?

We use our detection approach to detect occurrences of multi-language design smells following the methodology discussed in Section 9.2. For each file, we compute the value of a variable $Smelly_{i,r}$ that takes 1 if the file i contains at least one design smell in a specific release r,

[11] `http://mallet.cs.umass.edu/`

Table 9.4 Percentage of JNI Files Participating in Design Smells in the Release of 9 Systems

Systems	Releases Analyzed	% Files with Smells	Smells Density per KLOC
Zstd-jni	0.4.4 - latest release	61.36%	8.14
Javacpp	0.9 - 1.5.1-1	58.97%	17.84
Rocksdb	5.0.2 - latest release	36.30%	8.54
Java-smt	1.0.1 - 3.0.0	36.21%	26.08
VLC-android	3.0.0 - latest release	30.49%	17.67
Conscrypt	1.0.0.RC2 - 2.3.0	30.21%	14.05
Pljava	REL1_5_STABLE - latest release	30.13%	7.59
Realm	0.90.0 - 5.15.0	11.67%	4.63
Jpype	0.5.4.5 - latest release	7.45%	7.45
Average		**33.95%**	**12.44**

and 0 otherwise. We also compute $Native_{i,r}$ which takes 1 if the file i in a specific release r is native and 0 if not, following the rules discussed in Section 9.2.7. Since our tool is focusing on the combination of Java and C/C++, we compute for each release the percentage of files participating in at least one design smell out of the total of JNI files (files involved in Java and C/C++).

Table 9.4 summarises our results on the percentages of files with design smells in each of the studied systems. We report in this table the average number of JNI files participating in, at least one of the studied design smells for each system. Our results show that indeed, the multi-language design smells presented in Chapter 7 are prevalent in the nine studied open source projects with average occurrences from 10.18% in *Jpype* system to 61.36% in *Zstd-jni*. The percentage of files with smells differ from one project to another. We compute the average of the percentage of smells in all the systems. We find that on average, one-third (33.95%) of the JNI files in the studied systems contain multi-language design smells.

Besides analyzing in each system the percentage of files affected by each of the studied design smells, we also investigate their evolution over the releases. Figure 9.2 presents an overview of the evolution of the percentage of files participating in multi-language design smells in the releases of each system. All the details and data are available in the replication folder. The X-axis in Fig. 9.2 represents the releases analyzed. The Y-axis represents the percentage of files affected by at least one of the studied design smells, while the lines are related to each system. Results show that these percentages vary across releases in the nine systems with peaks as high as 69.04%. Some of these systems *i.e., Realm* and *Jpype* contain respectively 4.61% and 6.41% in the first releases, but the occurrences of smells increased over time to reach respectively 15.66% and 32.94%.

Overall, the number of occurrences of smells are increasing over the releases. Although, in some cases such as in *Rocksdb*, the number of occurrences seems to decrease from one release to the next one, (from 43.78% to 31.76%). The fact that developers might not be

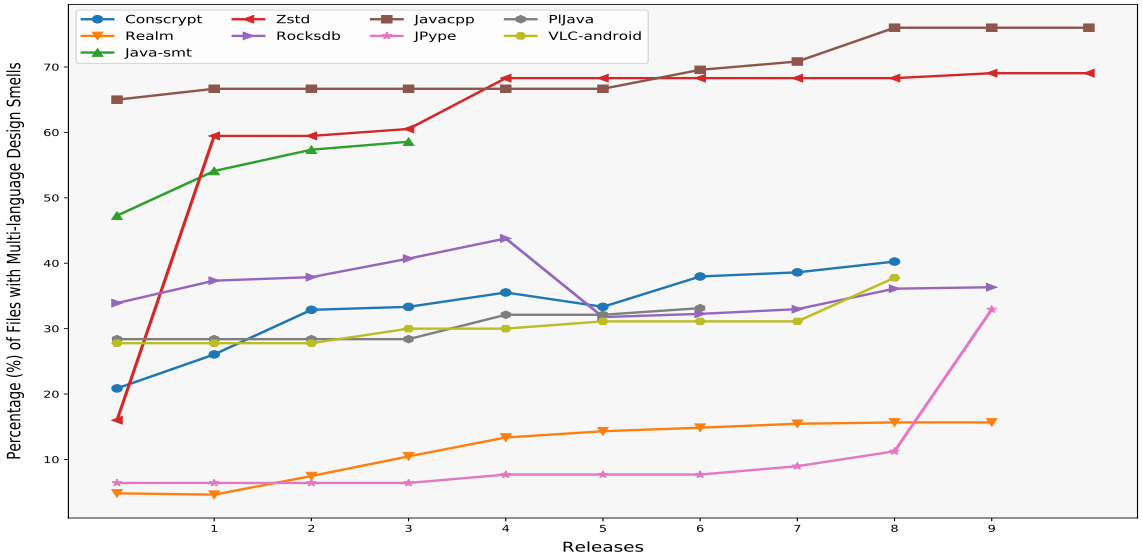


Figure 9.2 Evolution of Design Smells in the Releases of the 9 Systems

aware of occurrences of such smells and the lack of tools for their detection might explain the observed prevalence. The observed decrease in the number of occurrences observed in certain cases could be the result of fault-fixing activities, features updates, or any other refactoring activities. In general, as one can see in Fig. 9.2, these decreases are temporary; the number of occurrences often increase again in the next releases. Overall, the proportions of files with smells are considerably high and the smells persist, thus allowing to reject H_1.

Summary of findings (RQ9.1): The studied multi-language design smells are prevalent and persistent in open source projects. The number of their occurrences even increases over the releases.

RQ9.2: Are some specific multi-language design smells more frequent than others in open source projects?

Similar to **RQ9.1**, we use our approach from Chapter 8 to detect the occurrence of the 15 design smells in the nine subject systems. For each file and for each release, we defined a metric $Smelly_{i,r}$ which takes the value 1 if the release r of the file contains the design smell type i and 0 if it does not contain that specific smell. We compute for each release the number of files participating in that specific smell. Note that the same file may contain more than one smell.

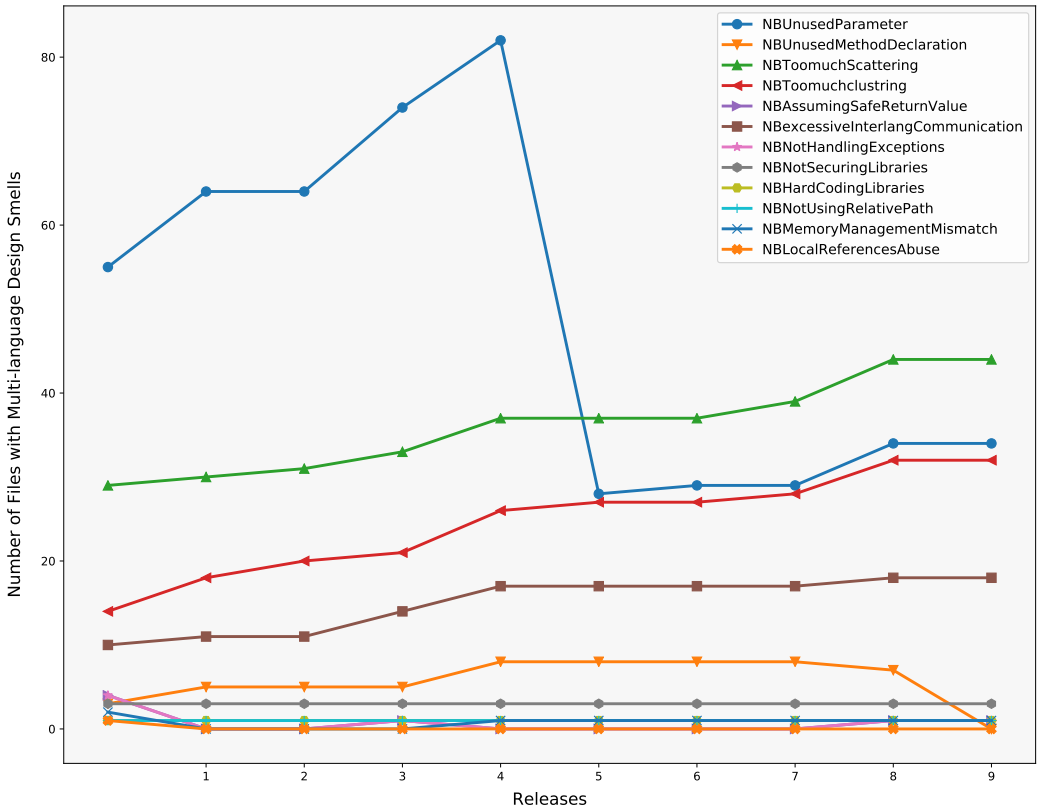


Figure 9.3 Evolution of the Different Kinds of Smells in *Rocksdb* Releases

Table 9.5 shows the distribution of the studied smells in the analyzed open source systems. We calculate the percentage of files containing these smells and compute the average. Since our goal is to investigate if some specific smells are more prevalent than others, we compute the percentage of files containing that specific smell out of all the files containing smells. Our results show that some smells are more prevalent than others, *i.e., Unused parameter, Too much scattering, Too much clustering, Unused Method Declaration, Not securing libraries, Excessive Inter-language communication.* In studied releases from *Jpype*, on average, 89.24% of the smelly files contain the smell *Unused parameter*. In *Java-smt*, on average, 94.06% of the smelly files contain the smell *Unused Parameters*. Our results also show that some smells discussed in the literature and developers' blogs have a low diffusion in the studied systems, *i.e., Excessive objects, Not caching objects, Local reference abuse*, while the other smells are quite diffused in the analyzed systems. *Conscrypt* presents 79.60% occurrences of the design smell *Unused Parameters*. As described in the commit messages in *Conscrypt*, this could be explained by the usage of BoringSSL which has many unused parameters. Results presented

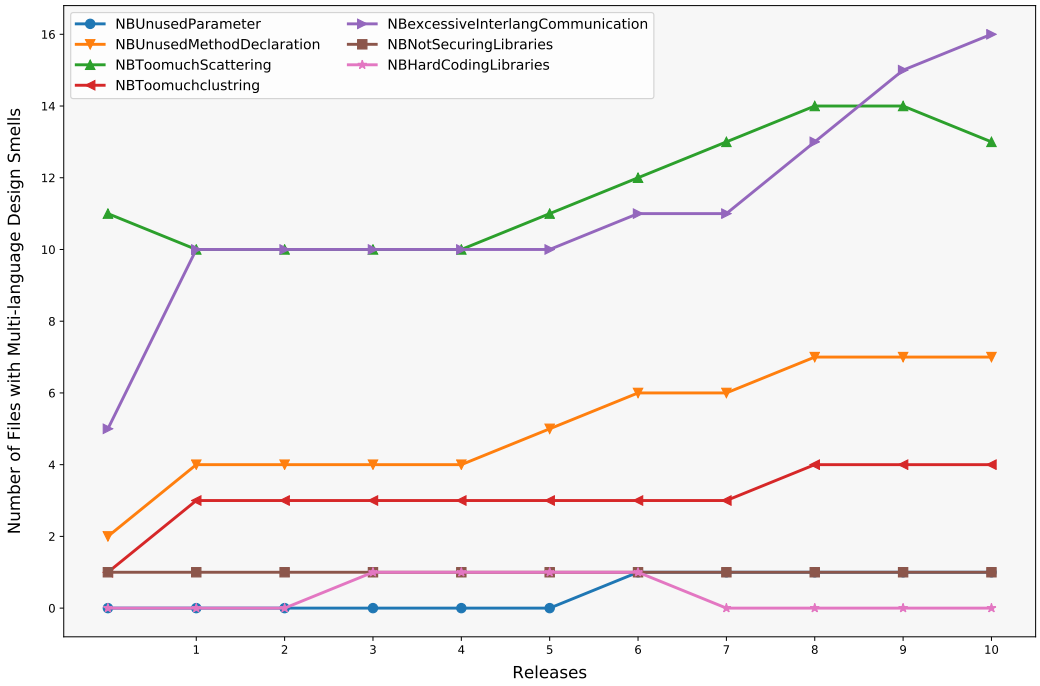


Figure 9.4 Evolution of the Different Kinds of Smells in *Javacpp* Releases

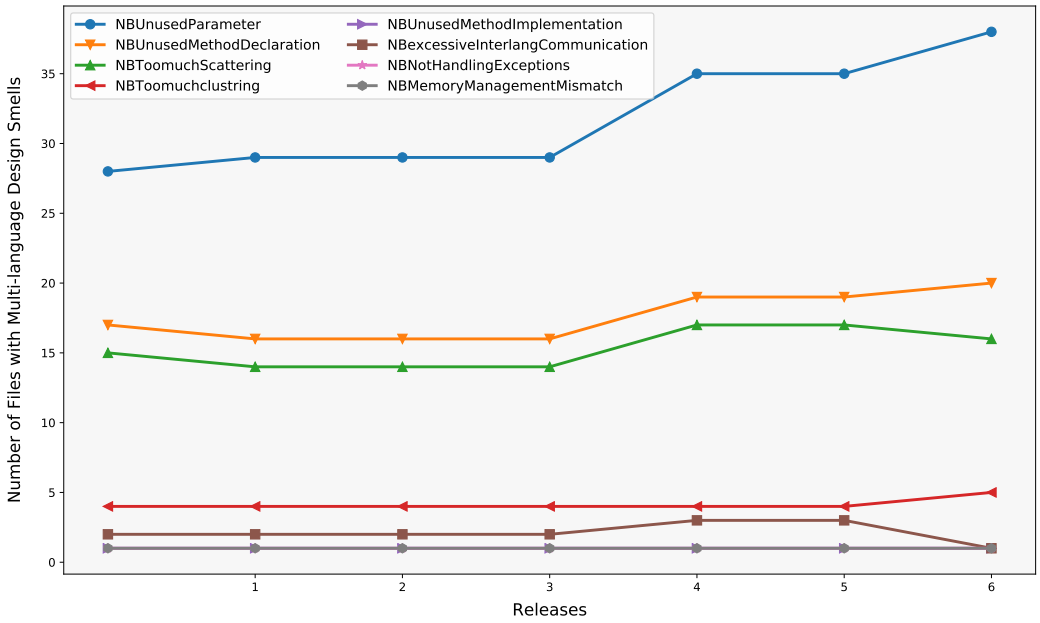


Figure 9.5 Evolution of the Different Kinds of Smells in *Pljava* Releases

Table 9.5 Percentage of JNI Files Participating in Design Smells in the Releases of the Studied Systems

System↓/Smells→	UP	UM	TMS	TMC	UMI	ASR	EO	EILC	NHE	NCO	NSL	HCD	NURP	MMM	LRA
Conscrypt	79.60%	4.40%	0%	1.90%	0%	3.99%	0%	1.90%	3.99%	0%	5.71%	0%	3.80%	3.78%	3.78%
Realm	67.68	3.066	9.75%	14.86	2.32%	4.33%	0%	12.58	5.15%	0%	2.17%	0%	0 %	0%	0.79%
Java-smt	94.06%	2.96%	0%	2.96%	0%	0%	0%	0%	0%	0%	2.96%	0%	2.96%	0%	0%
Zstd-jni	10.46	0.95%	13.98	12.36	3.47%	17.98	0%	23.55	21.45	0%	5.74%	3.47%	0%	2.25%	0%
Rocksdb	44.55%	5.48%	34.48%	23.47%	0%	0.67%	0%	14.35%	0.67%	0.91%	2.85%	0.95%	0.95%	0.79%	0.10%
Javacpp	2.53%	31.70	74.19	19.49	0%	0%	0%	69.14	0%	0%	6.48%	2.51%	0%	0%	0%
Jpype	89.24%	0%	0%	0%	0%	1.78%	0%	0.35%	1.78%	0%	0%	0%	0%	8.25%	1.07%
Pljava	64.45	35.62	31.02	8.42%	2.04%	0%	0%	4.36%	2.04%	0%	0%	0%	0%	2.04%	0%
VLC-android	63.67%	25.71%	24.74%	17.10%	7.34%	3.67%	0.82%	13.29%	3.67%	0%	3.92%	0%	6.01%	0%	3.67%
Median	64.45	4.4	13.98	12.36	0	1.78	0	12.58	2.04	0	2.96	0	0	0.79	0.1
Average	57.36	12.21	20.91	11.17	1.69	3.60	0.09	15.50	4.31	0.10	3.31	0.77	1.52	1.9	1.05

Acronyms: Up: UnusedParameter, **UM:** UnusedMethodDeclaration, **TMS:** ToomuchScattering
UMI: UnusedMethodImplementation, **ASR:** AssumingSafeReturnValue, **EO:** ExcessiveObjects
EILC:excessiveInterlangCommunication, **NHE:** NotHandlingExceptions, **NCO:** NotCachingObjects
HCD: HardCodingLibraries, **NURP:** NotUsingRelativePath, **MMM:** MemoryManagementMismatch
TMC: Toomuchclustring, **NSL:** NotSecuringLibraries **LRA:** LocalReferencesAbuse

in Table 9.5 report a range of occurrences from 0% to 94.06%. Some specific types of smells seem to be more frequent than others. On average *Unused Parameters* represents 57.36% of the existing smells, followed by the smell *Too Much Clustering* with 20.91%.

For each system, in addition to analyzing the percentage of files affected by each type of smell, we also investigate the evolution of the smell over the releases. Figures 9.3, 9.4, 9.5, 9.6, 9.7, and 9.8 provide an overview of the evolution of smells respectively in *Rocksdb*, *Javacpp*, *Pljava*, *Realm*, *Jpype*, and *Java-smt* releases. The X-axis in these figures represents the releases analyzed. The Y-axis represents the number of files in that specific system affected by that kind of design smells, while the lines are related to the different types of smells we studied. Depending on the system, some smells seem more prevalent than the others. In *Javacpp*, *Too Much Scattering*, and *Excessive Inter-language Communication* seem to be the predominant ones, while *Unused Parameters* is less frequent in this system. However, in general, for other systems including *Rocksdb* and *Realm*, *Unused Parameters* seems to be dominating. Results show that most of the smells generally persist within the project. The smells tend to persist in general or even increase from one release to another.

Although, in some specific cases, for example, the design smell *Unused Parameters* in *Rocksdb*, presented a peak of 82 and decreased to 28 in the next release. However, the number of files containing this smell increased in the next releases and reached to 34 in the last release analyzed. We studied the source code files containing some occurrences of the design smell *unused parameters* between releases (5.11.2 and 5.14.3) of *Rocksdb* to understand the reasons behind the peak and the decrease. We found that some method parameters were

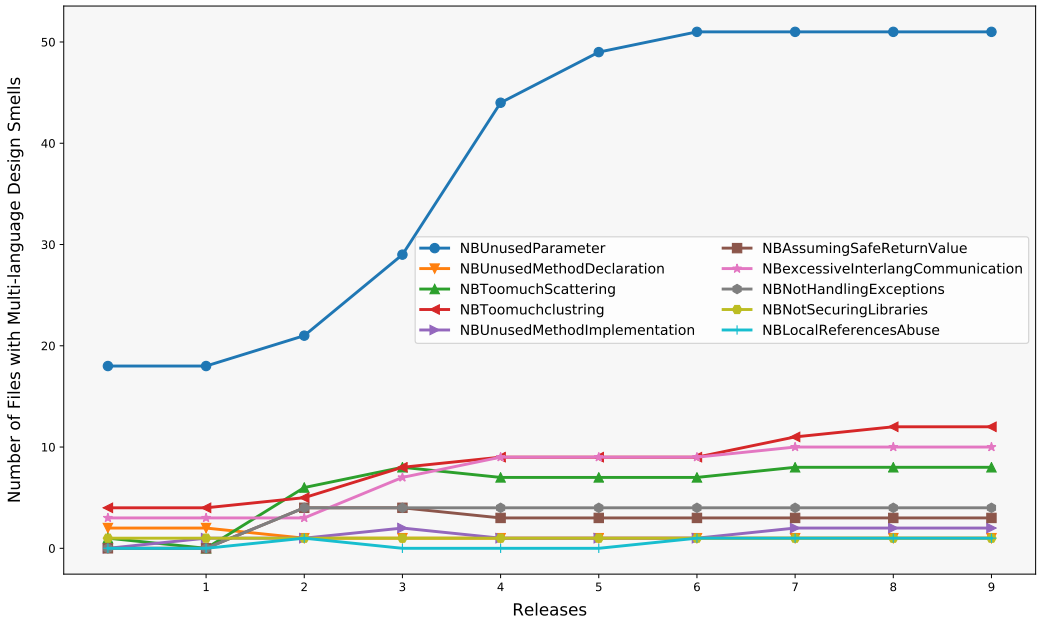


Figure 9.6 Evolution of the Different Kinds of Smells in *Realm* Releases

unused on *Rocksdb* (5.11.2) and have been refactored during the next releases by removing occurrences of this smell and also due to project migration features. Another example of refactoring of the code smell *Unused Parameters* from one release to another was observed in *Conscrypt*, where they refactored *Unused Parameters* occurrences due to errors generated by those occurrences in the release *1.0.0.RC14* (`"commit message: Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters"`). From our results, we can clearly observe that occurrences of design smells are not equally distributed. We conclude that the proportions of files with specific smells vary significantly between the different kinds of smells. We, therefore, reject hypothesis H_2.

Summary of findings (RQ9.2): Some design smells are more prevalent than others, *e.g., Unused Parameters*, *Too Much Scattering*, *Unused Method Declaration* while others are less prevalent, *e.g., Excessive Objects* and *Not Caching Objects*. Most of the smells persist with an increasing trend from one release to another in most of the systems.

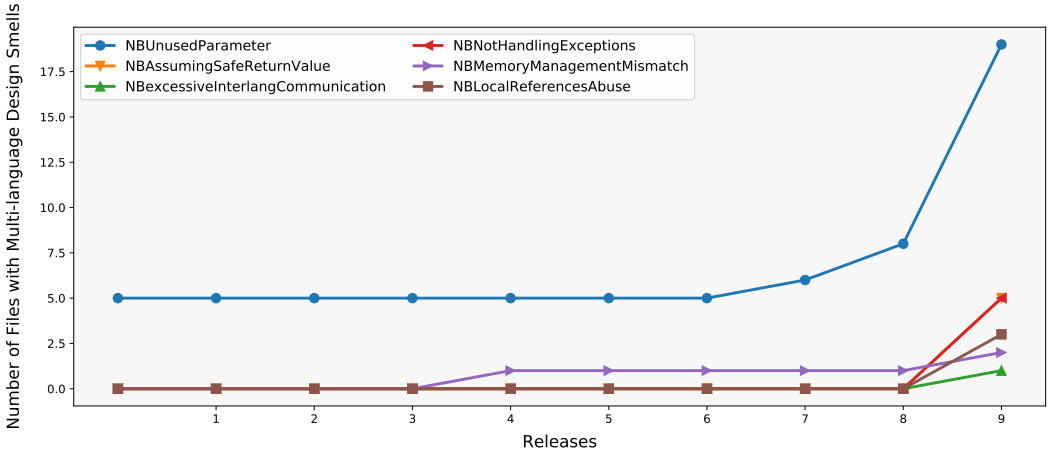


Figure 9.7 Evolution of the Different Kinds of Smells in *Jpype* Releases

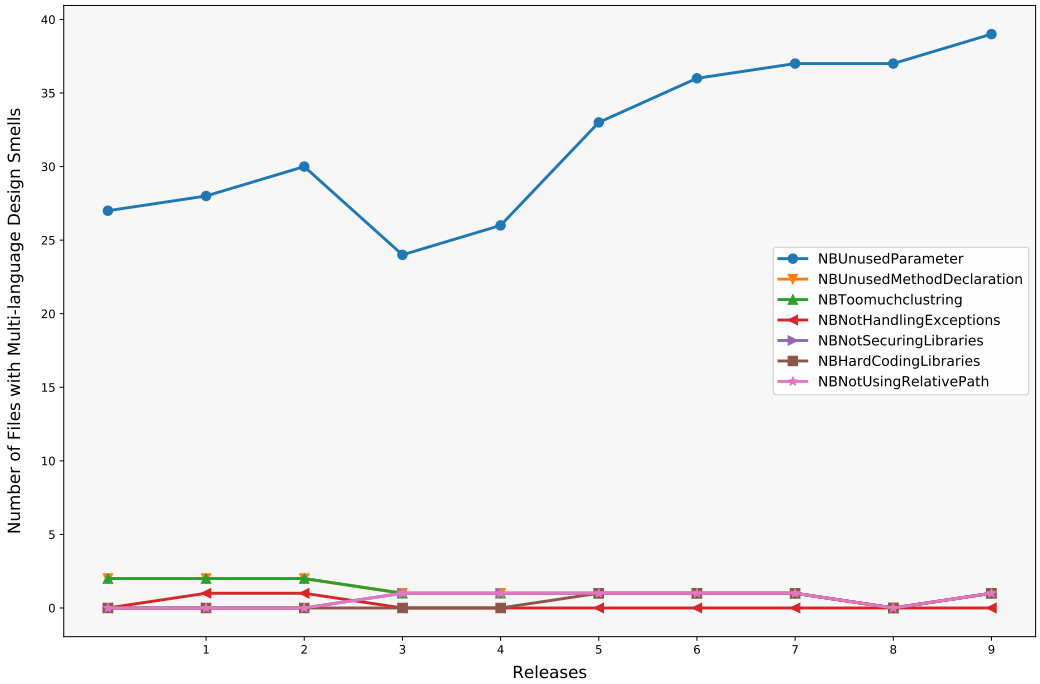


Figure 9.8 Evolution of the Different Kinds of Smells in *Java-smt* Releases

RQ9.3: Are files with multi-language design smells more fault-prone than files without?

Prior works show that design smells increase the fault-proneness of Java applications [23, 24]. Since JNI systems introduce other kinds of design smells and those smells are prevalent as observed in research questions RQ9.1 and RQ9.2, we are interested in studying the impacts of those design smells on the fault-proneness of JNI systems. For that, we applied Fisher's exact test [135] to check whether the proportion of bugs varies between two samples (files with and without smells) as discussed in Section 9.2.8. The columns *Smelly-buggy (SB), Buggy-Notsmelly (BNS), Smelly-NotBuggy (SNB), NotBuggy-NotSmelly (NBNS)* in Tables 9.6 and 9.7 contain the values of the contingency tables for the Fisher's exact test; each row corresponding to a single release. The numbers reported in the cells of these columns are the total number of JNI source code files (for the specific release) with or without smells and with or without bugs, depending on the column. More specifically, these columns present respectively: the total number of source code files with smells and are buggy (SB), the total number of source code files containing bugs without occurrences of smells (BNS), smelly source code files that are not buggy (SNB), and source code files that do not present any occurrences of smells or bugs (NBNS). The value of the odds ratio (OR) greater than 1 from Fisher's exact test indicates that files with design smells have higher odds of being buggy compared to files without design smells. The values $OR < 1$ indicate that files with design smells have lower odds of having faults, while $OR = 1$ refers to no impact of design smells on fault-proneness of the source files. The p-value shows the probability of observing the odds ratio by chance, and thus lower values (< 0.05) of p-value confirm the significance of the impacts of design smells on fault-proneness. In addition to significant p-values, we examine the confidence intervals of the odds ratios. A confidence interval specifies the range where the true odds ratio lies in. A significant p-value value (< 0.05) of an odds ratio (> 1.0) with confidence interval not containing 1 confirms a true relationship between design smells and fault-proneness. We marked the p-values of such cases with (*) in Table 9.6 and Table 9.7.

Tables 9.6 and 9.7 report the results of applying Fisher's exact test and present the values of odds ratios for the studied systems. Each row of those tables shows, for each system and each release, the odds for a file containing at least one type of design smells to be involved in a bug inducing change.

In most of the analyzed releases, Fisher's exact test indicates a significant difference of proportions between fault-prone JNI files with and without design smells. In some systems (*e.g., Rocksdb*, *Javacpp*, and *Java-smt*), odds ratios for specific releases are less than one, or the p-value is not statistically significant. However, in general, the values of odds ratios are high

Table 9.6 Fisher's Exact Test Results for the Fault-proneness of Files with and without Design Smells (1)

System	Releases	SB	BNS	SNB	NBNS	Odds Ratios	*p*-values	Confidence Interval
Rocksdb	rocksdb-5.0.2	82	85	17	108	6.1287	< **0.01***	(1.2184, 2.4076)
	rocksdb-5.4.6	89	90	23	98	4.2135	< **0.01**	(0.8979, 1.9787)
	rocksdb-5.6.2	90	80	24	107	5.0156	< **0.01***	(1.0771, 2.1480)
	rocksdb-5.9.2	97	84	30	101	3.8876	< **0.01**	(0.8563, 1.8592)
	rocksdb-5.11.2	99	86	42	95	2.6038	< **0.01**	(0.4929, 1.4211)
	rocksdb-5.14.3	50	101	38	88	1.1464	0.607	(-0.3729, 0.6462)
	rocksdb-5.17.2	51	92	39	97	1.3787	0.249	(-0.1840, 0.8263)
	rocksdb-5.18.3	50	101	43	88	1.0131	1.0	(-0.4848, 0.5109)
	rocksdb-6.1.1	49	94	55	90	0.8529	0.541	(-0.6404, 0.3224)
	rocksdb-latest release	49	101	56	83	0.7190	0.181	(-0.8108, 0.1511)
Pljava	pljava-1_4_3	0	36	0	100	-	1.0	-
	pljava-rel1_5_stable	33	38	13	78	5.2105	< **0.01**	(0.9008, 2.4005)
	pljava-1_5_0b3	32	33	14	83	5.7489	< **0.01***	(1.0026, 2.4954)
	pljava-1_5_0	33	37	13	79	5.4199	< **0.01**	(0.9388, 2.4413)
	pljava-1_5_1b1	32	38	14	78	4.6917	< **0.01**	(0.8077, 2.2839)
	pljava-1_5_1b2	39	36	14	76	5.8809	< **0.01***	(1.0436, 2.4998)
	pljava-1_5_2	38	34	15	78	5.8117	< **0.01***	(1.0392, 2.4806)
	pljava-latest release	39	35	16	76	5.2928	< **0.01**	(0.9600, 2.3727)
Realm	realm-java-0.90.0	21	89	2	365	43.0617	< **0.01***	(2.2938, 5.2315)
	realm-java-1.2.0	20	169	2	285	16.8639	< **0.01***	(1.3592, 4.2912)
	realm-java-2.3.2	33	177	3	269	16.7175	< **0.01***	(1.6194, 4.0135)
	realm-java-3.7.2	43	165	8	271	8.8280	< **0.01***	(1.3988, 2.9570)
	realm-java-4.4.0	48	166	18	262	4.2088	< **0.01**	(0.8616, 2.0127)
	realm-java-5.4.0	50	165	21	261	3.7662	< **0.01**	(0.7804, 1.8718)
	realm-java-5.7.1	52	164	22	260	3.7472	< **0.01**	(0.7856, 1.8565)
	realm-java-5.9.0	54	161	23	260	3.7915	< **0.01**	(0.8066, 1.8589)
	realm-java-5.11.0	54	161	24	259	3.6195	< **0.01**	(0.7668, 1.8059)
	realm-java-5.15.0	54	162	24	258	3.5833	< **0.01**	(0.7569, 1.7957)
VLC-android	vlc-android-3.0.92	19	23	8	40	4.1304	< **0.01**	(0.4460, 2.3907)
	vlc-android-3.1.6	22	22	6	40	6.6666	< **0.01**	(0.8552, 2.9390)
	vlc-android-3.1.0	22	22	6	40	6.6666	< **0.01**	(0.8552, 2.9390)
	vlc-android-3.0.13	18	24	7	41	4.3928	< **0.01**	(0.4720, 2.4879)
	vlc-android-latest release	21	23	13	33	2.3177	0.081	(-0.0322, 1.7134)
	vlc-android-3.0.11	19	24	6	41	5.4097	< **0.01**	(0.6411, 2.7352)
	vlc-android-3.0.0	19	22	6	43	6.1893	< **0.01**	(0.7709, 2.8747)
	vlc-android-3.0.96	19	23	8	40	4.1304	< **0.01**	(0.4460, 2.3907)
	vlc-android-3.1.2	22	22	6	40	6.6666	< **0.01**	(0.8552, 2.9390)
Jpype	jpype-0.5.4.5	5	28	0	45	-	< **0.02**	-
	jpype-0.5.5.1	5	28	0	45	-	< **0.02**	-
	jpype-0.5.5.4	5	28	0	45	-	< **0.02**	-
	jpype-0.5.6	5	29	0	44	-	< **0.02**	-
	jpype-0.5.7	6	28	0	44	-	< **0.01**	-
	jpype-0.6.0	6	28	0	44	-	< **0.01**	-
	jpype-0.6.1	6	28	0	44	-	< **0.01**	-
	jpype-0.6.2	6	28	1	43	9.2142	< **0.05**	(0.0509, 4.3906)
	jpype-0.6.3	6	28	3	43	3.0714	0.158	(-0.3432, 2.5875)
	jpype-latest release	23	42	5	15	1.6428	0.430	(-0.6362, 1.6291)

* = significant p-values for odd ratios with confidence intervals not containing 1

Table 9.7 Fisher's Exact Test Results for the Fault-proneness of Files with and without Design Smells (2)

System	Releases	SB	BNS	SNB	NBNS	Odds Ratios	*p*-values	Confidence Interval
Javacpp	javacpp-0.5	0	9	0	5	-	1.0	-
	javacpp-0.9	10	4	3	3	2.5	0.612	(-1.0599, 2.8926)
	javacpp-1.1	0	10	0	4	-	1.0	-
	javacpp-1.2	9	5	5	2	0.72	1.0	(-2.2994, 1.6424)
	javacpp-1.2.1	12	5	2	2	2.4	0.574	(-1.3449, 3.0958)
	javacpp-1.2.7	7	4	7	3	0.75	1.0	(-2.1148, 1.5395)
	javacpp-1.3	10	1	4	6	15.0	< **0.05**	(0.2942,5.1218)
	javacpp-1.3.2	11	6	3	1	0.6111	1.0	(-2.9646, 1.9797)
	javacpp-1.4	12	5	4	2	1.2	1.0	(-1.8101, 2.1747)
	javacpp-1.4.2	14	4	3	3	3.5	0.306	(-0.6955, 3.2011)
	javacpp-1.4.4	11	2	8	4	2.75	0.378	(-0.9147, 2.9379)
	javacpp-1.5	14	5	5	1	0.56	1.0	(-2.9573, 1.7977)
	javacpp-1.5.1-1	10	4	9	2	0.5555	0.660	(-2.5093, 1.3337)
Zstd-jni	zstd-jni-0.4.4	4	0	0	21	-	< **0.01**	-
	zstd-jni-1.3.0-1	13	0	9	15	-	< **0.01**	-
	zstd-jni-1.3.2-2	15	2	7	13	13.9285	< **0.01**	(0.8958, 4.3721)
	zstd-jni-1.3.3-1	16	2	7	13	14.8571	< **0.01**	(0.9649, 4.4320)
	zstd-jni-1.3.4-1	20	1	8	12	30.0	< **0.01***	(1.2025, 5.5998
	zstd-jni-1.3.4-8	20	1	8	12	30.0	< **0.01***	(1.2025, 5.5998)
	zstd-jni-1.3.5-3	20	1	8	12	30.0	< **0.01***	(1.2026, 5.5999)
	zstd-jni-1.3.7	20	1	8	12	30.0	< **0.01***	(1.2026, 5.5998)
	zstd-jni-1.3.8-1	20	1	8	12	30.0	< **0.01***	(1.2026, 5.5998)
	zstd-jni-1.4.0-1	22	1	7	12	37.7142	< **0.01***	(1.4198, 5.8403)
	zstd-jni-latest release	22	1	7	12	37.7142	< **0.01***	(1.4198, 5.8403)
Conscrypt	conscrypt-1.1.1	42	52	12	46	3.0961	< **0.01**	(0.3758, 1.8844)
	conscrypt-1.0.0.RC14	4	64	0	54	-	0.128	-
	conscrypt-1.0.1	38	55	11	43	2.7008	< **0.02**	(0.2128, 1.7743)
	conscrypt-2.1.0	47	53	17	42	2.1908	< **0.05**	(0.0975, 1.4711)
	conscrypt-1.0.2	38	55	11	43	2.7008	< **0.02**	(0.2128, 1.7742)
	conscrypt-1.4.2	6	0	55	97	-	< **0.01**	-
	conscrypt-1.2.0	45	52	15	46	2.6538	< **0.01**	(0.2697, 1.6823)
	conscrypt-1.0.0.RC11	37	55	11	43	2.6297	< **0.02**	(0.1844, 1.7493)
	conscrypt-1.0.0.RC2	23	20	6	90	17.25	< **0.01***	(1.8270, 3.8686)
	conscrypt-1.0.0.RC8	26	59	11	46	1.8428	0.172	(-0.1922, 1.4148)
Java-smt	java-smt-0.60	0	23	0	7	-	1.0	-
	java-smt-1.0.1	21	20	5	9	1.89	0.3667	(-0.6165, 1.8896)
	java-smt-2.0.0-alpha	22	16	11	12	1.5	0.5966	(-0.6357, 1.4467)
	java-smt-2.2.0	30	19	9	10	1.7543	0.4132	(-0.5061, 1.6304)
	java-smt-3.0.0	19	17	22	12	0.6096	0.3414	(-1.4556, 0.4658)

* = significant p-values for odd ratios with confidence intervals not containing 1

(in general greater than 2) in most cases. For *Zstd-jni*, we found odds ratios always higher than 13. Having this high odds could be explained by the large number of smells contained in that system, as described in Table 9.4, but also by the nature of smells existing in this system. The higher values of statistically significant odds ratios in most cases and the confidence intervals of those significant odds ratios being above the value 1 in some cases show that multi-language design smells are related to fault-proneness. However, this relationship varies with systems and further investigation is necessary to generalize.

From analyzing fault-fixing commit messages, we identified some commits reporting a refactoring for specific smells *e.g.,* `"removing unused parameter", "implementing the handling of exception"`. This could explain cases where *Smelly-Buggy* values decrease from one re-

lease to the other, while the overall number of occurrences of smells are in general increasing from one release to the other, as shown in Fig. 9.2. For example, in *Rocksdb*, *Smelly-Buggy* values are decreasing from one release to the other, while *Smelly-NonBuggy* is increasing. This could be explained by the nature of the smells and the refactoring applied. Since one file can contain more than one type of smell, the refactoring of some specific types of smells could decrease the risk of bugs while leaving the file still smelly. This suggests that some specific smells could be more correlated with bugs than others. These hypotheses motivate us to further investigate the relationship between specific types of smells and fault-proneness (**RQ9.4**) and also the activities that once performed in smelly files could lead to bugs (**RQ9.5**).

We, therefore, conclude that, in most cases, there is a relation between multi-language design smells and fault-proneness in the context of JNI systems: a greater proportion of JNI files participating in design smells experienced bugs compared to other classes. We therefore reject H_3. The rejection of H_3 and the statistically significant odds ratios provide a *posteriori* concrete evidence of the impact of multi-language design smells on fault-proneness in the context of JNI files.

Summary of findings (RQ9.3): Our results suggest that files with occurrences of the studied smells are more likely to be associated with faults than files without these smells and this relationship is statistically significant in most cases.

RQ9.4: Are some specific multi-language design smells more fault-prone than others?

Findings from RQ9.3 suggest that source code files with smells in JNI systems are often more prone to faults than files without smells. Although these findings give a general impression of the impacts of smells on the fault-proneness of JNI systems, it is important to know which smell(s) are more related to faults. When we are able to identify some specific smells to be more related to faults, we can prioritize those smells during the maintenance of the JNI systems. As presented in our methodology described in Section 9.2.8, we apply multivariate logistic regression to examine whether some types of design smells are more related to fault-proneness. In our logistic regression models, independent variables are the number of occurrences of each type of design smells. The dependent variable is a dichotomous flag (*buggy*) that assumes values either 0 (non-buggy) or 1 (buggy). For each system, we build a logistic regression model and analyze the model coefficients and p-values for individual types of smells. To address multicollinearity among the independent variables, we drop one of the

variables from each highly correlated pair of variables from the models. From our analysis, we observed two pairs of smells highly correlated- (*Not Handling Exceptions, Assuming Safe Return Value*) and (*Not Securing Libraries, Not Using Relative Path*) with correlation (Spearman's) coefficients of 0.91 and 0.60, respectively. We keep *Not Handling Exceptions* and *Not Securing Libraries* as they are more prevalent in the systems compared to the other smell in each correlated pair. Similarly, we drop the variable code churn from the model as we found it to be highly correlated (0.99) with the file size (LOC). We chose Spearman's rank correlation as it is non-parametric and does not require data to be normally distributed.

We rank the independent variables based on the logistic regression model coefficients (log odds) and the corresponding p-values. Table 9.8 presents the model coefficients and their ranking for each system. The coefficients with significant p-values (<0.01) are presented in boldface. To evaluate the relationships of individual types of smells with fault-proneness, we summarize the data from Table 9.8 into Table 9.10 to identify the top five smell types that are more related to bugs. For each smell type, Table 9.10 presents the percentage of systems where the smell type has positive log odds, the number of times the smell type is in the top five in the ranking of positive log odds, and the number of systems where the log odds are statistically significant. For each smell, we consider only the system where we are able to calculate the model coefficients and thus we exclude the systems where we do not get the coefficient values due to singularities. For the smell *Too Much Clustering* for example, in Table 9.8, for eight (8) out of nine (9) systems (*i.e.,* except *Jpype*) we have values for model coefficients.

Out of these eight systems, for five (5) systems (*Conscrypt*, *Javacpp*, *Rocksdb*, *VLC-android*, and *Zstd-jni* *i.e.,* 5/8 (62.5%) times) *Too Much Clustering* has positive values for log odds. All of these five times (systems) the log odds were ranked in the top five, having significant p-values in four systems (*Conscrypt*, *Javacpp*, *VLC-android*, and *Zstd-jni*). For all smell types in Table 9.8, we present such summary in Table 9.10. We then report the top five smell types (underlined in Table 9.10) based on the percentage of systems in which the smell have positive log odds, number of times the positive log odds were in the top five ranking, and the number of systems in which the log odds have significant p-values respectively as shown in Table 9.10. For the control variables LOC and the number of previous bug-fix, we observed positive log odds in most of the systems. So, these coefficients for the control factors agree with the known impacts of these two variables on fault-proneness. We also observed negative log odds for the smells from the logistic regression models for the studied systems. The negative regression coefficients might be interpreted as an indication that the corresponding smells are negatively related to fault-proneness. However, this scenario varies across the studied systems.

Table 9.8 Log Likelihood of Different Smells from the Logistic Regression models for Fault-proneness of the Studied Systems

Smells ↓ / Systems →	*Conscrypt*		*Java-smt*		*Javacpp*		*Jpype*		*Pljava*		*Realm*		*Rocksdb*		*VLC-android*		*Zstd-jni*	
	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank	Coeff.	Rank
EO	**-9.538e+15**		NA		**-1.765e+14**		-5.380e+01		-4.649e+01		4.495	2	1.640e+01	4	**-7.653e+13**		**-4.265e+13**	
TMC	**3.280e+15**	1	-6.864e+03		**3.280e+15**	1	NA		-5.488e+02		-9.514e+01		8.593e+02	3	**1.265e+15**	2	**2.949e+14**	3
TMS	NA		NA		**2.304e+14**	4	NA		1.492e+02	1	9.310e+01	1	1.324e+03	2	**3.438e+15**	1	**1.077e+15**	2
UMD	**-6.275e+13**		2.846e+01	1	**9.799e+13**	6	NA		4.883e+01	2	-6.092e+01		-6.785e+01		**-6.359e+12**		**-2.448e+13**	
UMI	NA		NA		NA		NA		-3.446e+02		-2.288e+02		NA		**-5.237e+14**		**9.511e+13**	5
UP	**8.145e+13**	3	8.497	2	**7.421e+14**	2	-5.527e-02		-7.090e-02		4.828	3	-2.286e+01		**3.363e+13**	3	**2.092e+13**	7
EO	NA		NA		NA		NA		NA		NA		NA		NA		NA	
NHE	**2.089e+14**	2	NA		NA		1.993e+01	1	-1.999e+02		-9.445e+01		-4.270e+01		**-2.951e+15**		**2.373e+14**	4
NCO	NA		NA		NA		NA		NA		NA		NA		NA		NA	
NSL	**-2.915e+13**		1.093	3	**3.308e+14**	3	NA		NA		-9.201e+01		-2.498e+03		**-8.185e+14**		**-1.569e+15**	
HCL	NA		-8.665e+01		**1.235e+14**	5	NA		NA		NA		4.936e+03	1	NA		**2.135e+15**	1
MMM	**-1.180e+15**		NA		NA		1.162e-01	2	NA		NA		-1.393e+03		NA		**4.684e+13**	6
LRA	**-1.301e+15**		NA		NA		-2.462e+02		NA		-1.891e+02		-3.474e+04		**-8.620e+15**		NA	
LOC	**-8.459e+11**		-4.280e-03		**1.159e+11**		8.468e-05		5.328e-04		2.488e-03		8.713e-01		**1.091e+11**		**5.331e+10**	
Prev. bug-fix	**6.363e+14**		1.864e+02		**1.293e+15**		5.728e+01		3.010e+02		9.480e+01		1.448e+03		**7.873e+14**		**7.437e+14**	
Null deviance	2034.3		5.2010e+02		349.15		1.0902e+03		2.3975e+03		6.6378e+03		4048.24		1245.10		589.62	
Residual deviance	1513.8		2.7256e-09		2595.14		3.5622e-10		4.8079e-10		8.8774e-10		8.5475		360.44		2595.14	
AIC	1535.8		16		2615.1		16		20		24		34.55		384.44		1107.3	

Table 9.9 Fault-proneness of Different Types of Smells Based on Logistic Regression Model for All Systems

Smell Types	Log Odds	p-values
Excessive Inter-language Communication	2.985e-01	1.11e-07 (<0.01)
Too Much Clustering	4.262e+00	0.000898 (<0.01)
Too Much Scattering	8.359e+00	1.82e-15 (<0.01)
Unused Method Declaration	9.078e-01	< 2e-16 (<0.01)
Unused Method Implementation	-1.255e+01	0.894915
Unused Parameters	5.695e-01	< 2e-16 (<0.01)
Not Handling Exceptions	3.248e+00	0.000217 (<0.01)
Not Securing Libraries	-4.469e-01	0.813090
Hard Coding Libraries	1.841e+00	0.546194
Memory Management Mismatch	-9.255e+00	0.998258
Local References Abuse	-1.172e+01	0.999968
Excessive Objects	NA	NA
Not Caching Objects	NA	NA

NA = Corresponding Log odds are not available from the LR models due to singularities

As shown in Table 9.8, the log odds of the independent variables vary across the systems. In four of the systems (*Conscrypt*, *Javacpp*, *VLC-android*, and *Zstd-jni*), we observe that the log odds for the smells are statistically significant (<0.01). These four systems reject the hypothesis H_4, meaning that different smells have different impacts on fault-proneness. However, we cannot generalize it to other systems to have a concrete conclusion. Thus, given the varying log odds of the smells from our regression models for individual systems, we conclude that the relationships between different types of multi-language smells and fault-proneness are system dependent.

Given that we have limited evidence to draw a firm conclusion on the strength of the relationships between different types of smells and fault-proneness, we focus on identifying smells that are relatively more related to faults based on the ranking of the values of the log odds and their significance. In Table 9.10, the smell type *Too Much Clustering* has positive log odds in 62.5% (5/8) of the systems. Each time, log odds were among the top 5 and was statistically significant in three systems. Similarly, *Too Much Scattering*, *Unused Parameters*, *Hard Coding Libraries* and *Memory Management Mismatch* are among the top five smells with positive log odds in 100%(6/6), 66.6%(6/9), 75%(3/4), and 50%(2/4) systems, respectively. These smells are likely to have a strong relation with fault-proneness. Besides, the smells *Not Handling Exceptions*, *Unused Method Declaration* and *Not Securing Libraries* have significant positive log odds for 2 (*Conscrypt*, *Zstd-jni*), 1 (*Javacpp*), and 1 (*Javacpp*) system(s), respectively; indicating some degree of relation with fault-proneness.

Table 9.10 Fault-proneness of Different Types of Smells Based on Logistic Regression Analysis

Smell Types	Number and Percentage of Systems		
	LO > 0	LO in Top 5	(LO>0 and p<0.01)
Excessive Inter-language Communication	25%(2/8)	2	0
Too Much Clustering	62.5%(5/8)	5	4
Too Much Scattering	100%(6/6)	6	3
Unused Method Declaration	37.5%(3/8)	2	1
Unused Method Implementation	25%(1/4)	1	1
Unused Parameters	66.6%(6/9)	5	4
Not Handling Exceptions	42.8%(3/7)	3	2
Not Securing Libraries	28.5%(2/7)	2	1
Hard Coding Libraries	75%(3/4)	3	2
Memory Management Mismatch	50%(2/4)	1	1
Local References Abuse	0%(0/5)	0	0
Excessive Objects	NA	NA	NA
Not Caching Objects	NA	NA	NA

LO = Log Odds of the corresponding smell from the logistic regression model.
NA = Corresponding Log odds are not available from the LR models due to singularities

Excessive Inter-language Communication and *Local References Abuse* which have no significant positive log odds are less likely to be associated with faults.

We also build a single logistic regression model for all the systems combined to evaluate how the findings from individual systems generalize. We presented the regression results for the smells in Table 9.9. We observed that smells *Excessive Inter-language Communication*, *Too Much Clustering*, *Too Much Scattering*, *Unused Method Declaration*, *Unused Parameters*, and *Not Handling Exceptions* have positive log odds with significant p-values (<0.01). This is an indication that these smells have statistically significant relationships with fault-proneness. This finding corroborates our findings from the analysis of individual systems for most cases. However, we did not observe significant relationships between multi-language smells and fault-proneness for the remaining smell types (in the models for all systems combined). Now, if we consider positive log odds with significant p-values in the logistic regression model for all systems combined (Table 9.9) and the percentage of positive log odds for regression models for individual systems (Table 9.10), we observe that the smell types *Too Much Clustering*, *Too Much Scattering*, *Unused Parameters*, *Not Handling Exceptions*, and *Hard Coding Libraries* are the most related to fault-proneness. However, this relationship varies with systems. One important point to note is the fact that smells suggested by our empirical results to be more related to fault-proneness constitute roughly over 80% of the smells in the studied systems (details in Table 9.5). This further shows that it is important to detect and remove these smells from the systems as soon as possible.

To study the relationships between multi-language smells and fault-proneness, we also investigate the correlation (Spearman's) between the number of smells of individual types in a file and the number of bugs associated with the corresponding file. We observed that *Not Using Relative Path* (0.48), *Not Handling Exceptions* (0.32), *Excessive Inter-language Communication* (0.30), *Local References Abuse* (0.24), *Hard Coding Libraries* (0.19), and *Too Much Clustering* (0.18) are the top smells based on the correlation with faults, although most of these and the remaining correlations are weak. Also, we mentioned that *Not Using Relative Path* was dropped from our logistic regression models because of its high correlation with *Not Securing Libraries*. So, we cannot draw a firm conclusion on the impacts of smells on fault-proneness based on these correlation results.

To have better insights into the identified relationships between the different types of design smells and bugs and to understand the bug-smell contexts in the studied systems, we further manually investigated a random sample of commit messages associated with bugs. From analyzing these commit messages, we found some commit messages clearly suggesting that some specific smells are often related to bugs. For example, in the release *1.0.0.RC14* of *Conscrypt*, a commit message is clearly specifying errors related to the code smell *Unused Parameters* (`"Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters"`). The same goes for *Memory Management Mismatch*, in *Realm*, a commit message was discussing errors related to memory management `"DeleteLocalRef when the ref is created in loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it"`. Another example from *Conscrypt* discussing bugs related to native memory management `"This fixes a memory leak in NativeCrypto_i2d_PKCS7. It never frees derBytes"`. The smell *Not Handling Exceptions* is also discussed as related to the bug 3482 in *Realm* (`"Add cause to RealmMigration Needed Exception (#3482)"`). *VLC-android* also presents bugs related to the smell *Not Handling Exceptions* `"rework exceptions throwing from jni"`. Similarly, other commit messages were also describing bugs related to *Unused Method Declaration*. `"There were a bunch of exceptions that are being thrown from JNI methods that aren't currently declared"`, and `"Fix latent bug in unused method"` present examples extracted respectively from *Conscrypt* and *Pljava*. Thus, we see that our identified smells are often related to bugs which highlight practical contexts and usability of our findings.

We also analyzed some quality attributes of our models such as *Null deviance, Residual deviance, and Akaike's Information Criterion (AIC)* as presented in Table 9.8. We observe that there are larger differences between null deviance and the residual deviance for the models of all the systems indicating a good fit of the regression models. We observe lower

values for AIC for most of the regression models indicating the simplicity of the models with comparatively higher values for *Conscrypt*, *Javacpp*, *VLC-android*, and *Zstd-jni*.

Summary of findings (RQ9.4): We conclude that although not always significant, there exists a relation between types of smells and the fault-proneness. The relationship is not consistent for all types of smell and across all the systems. Smell types *Too much Scattering, Too Much Clustering, Unused Parameters, Hard Coding Libraries, and Not Handling Exceptions* are observed to be more related to faults compared to other smells, and thus should be prioritized during maintenance.

RQ9.5: What are the activities that are more likely to introduce bugs in smelly files?

Since the risk of having bugs could differ from one activity to the other, we decided to investigate what kind of activities once performed in smelly files could increase the risk of bug occurrences. Having knowledge of risky activities developers and maintainers could reduce the risk of bugs in smelly files. To study the activities that could introduce bugs in smelly files, we collected the fault-inducing commits messages and performed a topic modeling, combining a mix of manual and automatic approaches as described in Section 9.2.

Table 9.11 lists 12 activities that are more likely to introduce bugs in smelly files. For each activity the table lists the systems from which the activity was extracted. For each activity, we also present examples of keywords used to build the topic for that activity. For example, the activity memory management, was extracted using a set of keywords including: buffer, memory, leak, flush, reference, local, memtable from *Rocksdb*, *Realm*, *Conscrypt*, and *Jpype*. `"Add more numbers to float-conversion test, add new unit-test for float-conversion"`, is an example of a commit message describing data conversion activity when the bug was introduced, extracted from *Java-smt*. Another example of commit message that introduced bugs extracted from *Rocksdb*: `"Another change is to only use class BlockContents for compressed block, and narrow the class Block to only be used for uncompressed blocks, including blocks in compressed block cache"`. This commit message is related to compression activities. From *Zstd-jni*, the following commit messages `"expose faster API to allow re-using of dictionaries"` refers to the usage of APIs. Those activities were extracted from commit messages of fault-inducing commits. Developers performed those activities in files containing occurrences of multi-language design smells when the bug was introduced.

Table 9.11 Activities Introducing Bugs in Smelly Files

No	Activities	Example of Keywords	Systems
1	Compression tasks	compression, decode, memory, blocks, encode, compaction, streaming, frames, block, dictionary.	*Rocksdb*, *Zstd-jni*
2	Data conversion	parser, type, container, basic, declaration, write, invalid, convert, string, coverage.	*Rocksdb*, *Realm*, *Pljava*, *Javacpp*, *Conscrypt*, *Java-smt*, *Zstd-jni*
3	Memory management	buffer, messagesize, memory, leak, local, reference, flush, memtable, allocation, garbage.	*Rocksdb*, *Realm*, *Conscrypt*, *Pljava*, *Jpype*, *Javacpp*
4	Restructuring the code	add, update, remove, code, reorder, native, move, public, improve, change.	*Rocksdb*, *VLC-android*, *Conscrypt*, *Jpype*, *Pljava*
5	Database management	stored, db, database, persistence, key, data, visible, size, file, timestamp.	*Rocksdb*, *Pljava*
6	API usage	external, library, api, include, expose, public, integrate, allow, streaming, wrapper.	*VLC-android*, *Realm*, *Conscrypt*, *Rocksdb*, *Pljava*, *Zstd-jni*,*Java-smt*
7	Feature migration	upgrade, support, migrate, integrate, create, legacy, simplify, add, format, update.	*Realm*, *Javacpp*, *Rocksdb*, *Java-smt*, *Zstd-jni*
8	Network management	sslsocket, encrypt, socket, nativessl, token, hostname, protocol, platform, sslSession, activesession.	*Conscrypt*, *Rocksdb*
9	Exception management	occur, handle, check, exception, throw, return, fix, pointer, illegal, runtime.	*Conscrypt*, *Javacpp*, *Jpype*, *Java-smt*
10	Threads management	thread, pull, execution, reflect, client, transaction, monitor, notify, mutex, log.	*Pljava*, *Rocksdb*, *Realm*
11	Performance management	time, wrap, performance, execution-time, regression, cache, shared, resources, bundle, increase.	*Zstd-jni*, *Rocksdb*
12	Compiler management	compiler, resolve, failure, check, warnings, support, JNI_ABORT, error, illegal, dynamic.	*Jpype*, *Rocksdb*

From analyzing the commit messages and topics of activities, we found that activities related to data conversion, memory management, API usage, code restructuring, and exception management are the most common activities that could increase the risk of bugs when performed in smelly files. Activities related to the compiler management, threads management, and compression tasks could also induce bugs in smelly files.

To understand why these activities seem to be risky, we decided to investigate further these activities at the source code level. For example, zstd.java is a native class from *Zstd-jni* system. This class contains 1351 lines of code with 75 native methods and exhibits the smell *Too Much Clustering*. This class combines methods performing distinct responsibilities, *i.e.,* compression, decompression, computation, data access, and utility methods. As per commit messages identified as introducing bugs, the developer was reordering the statics methods, adding JNI wrappers, and performing compression tasks `"Reorder the static methods, All compression first then all decompression then the rest, inputs checking + utility methods", "Add Java wrappers and C implementations of compress/ decompress using direct ByteBuffer"`. This class contains two types of smells, *Too Much Clustering* and *Excessive Inter-language Communication*. The nature of those two smells is by definition adding complexity to the code by making the readability of such classes hard. Thus, restructuring the code of a large native class could have increased the risk of introducing bugs. Indeed, applying changes on multi-language code could bring some confusion

if the developer is not familiar with the components involved in the multi-language interaction. Similarly, activities related to compression are declared in Java side and mainly implemented in the C/C++ side (jni_zstd.c). Developers should have knowledge of both implementations to correctly perform a change, especially that this class contains *Excessive Inter-language Communication* between Java and C/C++ when performing compression activities. Another example is illustrated by Listing 9.1. It presents a function extracted from the C file jni_zdict.c from *Zstd-jni* system. A developer was `"adding support for legacy dictionary trainer"` in this smelly function when the bug was introduced. However, in the context of JNI, it is important to always perform checks to ensure that the native execution was performed correctly. As described in Chapter 7, when checking JNI exceptions, we should add a return statement just after throwing the excepting to interrupt the execution flow and exit the method in case of errors. The `ThrowNew()` functions do not interrupt the control flow of the native method. In case an error occurred when retrieving the jclass, the exception will not be thrown in the JVM until the native method returns. Developers should be aware of how to implement the exception in the context of JNI systems to avoid introducing bugs related to mishandling JNI exceptions. Activities related to the conversion of types could also introduce bugs as expressed by a commit message extracted from *Javacpp* ; *i.e.,* `"Provide ‘BytePointer’ with value getters and setters for primitive types other than ‘byte’ to facilitate unaligned memory accesses"`.

Another example of bugs related to the management of the memory is extracted from *Pljava*, c source code file JNICalls.c, `"Eliminate threadlock ops in string conversion"`. Both of those files exhibit the smell *Memory Management Mismatch*. Activities related to data and type conversion could increase the risk of bug because when converting types from Java to C/C++, the conversion will raise two categories of types; primitive types and reference types. Primitive types are simple to convert, we usually add j in front of the type *e.g.,* `int` become `jint`, `float` become `jfloat`, etc. However, for the reference types *i.e.,* Class, Object, String, developers should use the predefined method to correctly perform the conversion. However, it happens that they forget to release the memory after such conversion which could introduce additional bugs including memory leaks. Listing 9.2 presents an example extracted from *Pljava* as introducing bugs. In this example, the method `GetObjectArrayElement` is used to capture a Java array. However, the memory is not released after usage as done in Listing 9.1. From the above examples, we conclude that some specific types of activities are relatively more frequently associated with bugs, especially in the context of multi-language design smells. Developers should be cautious while performing those activities.

Listing 9.1 Example of Bug in Smelly Method 1/2

```
/*
...
*/
 jsize num_samples = (*env)->GetArrayLength(env, sampleSizes);
    jint *sample_sizes_array = (*env)->GetIntArrayElements(env, sampleSizes, 0);
    size_t *samples_sizes = malloc(sizeof(size_t) * num_samples);
    if (!samples_sizes) {
        jclass eClass = (*env)->FindClass(env, "Ljava/lang/OutOfMemoryError;");
         (*env)->ThrowNew(env, eClass, "native heap");
    }
    for (int i = 0; i < num_samples; i++) {
        samples_sizes[i] = sample_sizes_array[i];
    }
   (*env)->ReleaseIntArrayElements(env, sampleSizes, sample_sizes_array, 0);
```

Summary of findings (RQ9.5): Activities related to data conversion, memory management, code restructuring, API usage, and exception management are the most common activities that could increase the risk of bugs once performed in smelly files, and thus should be performed carefully.

9.2.11 Discussion

This section discusses the results reported in Section 9.2.10.

9.2.12 Multi-language Design Smells

Distribution of design smells From our results we found that most of the studied smells specific to JNI systems are prevalent in the studied projects. Results from the studied systems reflect a range from 10.18% of smelly files in *Jpype* system to 61.36% of smelly files in *Zstd-jni*. On average, 33.95% of the JNI files in the studied systems contain multi-language design smells. Multi-language systems offer numerous benefits, but they also introduce additional challenges. Thus, it is expected to have new design smells specific to such systems due to their heterogeneity. The prevalence of multi-language smells in the studied projects highlights the need for empirical evaluation targeting the analysis of multi-language smells and also the study of their impact on software maintainability and reliability. We also analyzed the persistence of these smells. Our results show that overall the number of smells usually increases from one release to the other. Such systems usually involve several developers

Listing 9.2 Example of Bug in Smelly Method 2/2

```
/*
...
*/
   jsize idx;
   jboolean foundNull = JNI_FALSE;
   BEGIN_JAVA
   idx = (*env)->GetArrayLength(env, array);
   while(--idx >= 0)
   {    if((*env)->GetObjectArrayElement(env, array, idx) != 0)
         continue;
      foundNull = JNI_TRUE;
      break;
   }
   END_JAVA
   return foundNull;
}
```

working in the same team and who might not have a good understanding of the architecture of the whole project. Thus, the number of smells may increase if no tools are available to detect those smells and-or to propose refactored solutions.

We observed situations in which the number of smells could decrease from one release to the next one. From investigating the commit message, we observed that some smells were refactored from one release to the other. Most of them due to the side effect of other refactoring activities, but also due to specific refactoring activities, *e.g.,* removing *Unused Parameters*, unused methods, implementing the handling of native exceptions, etc. This suggests that some developers might be aware of the necessity to remove those smells. However, since no tools are available to automatically detect such occurrences, it is hard for a developer to manually identify all the occurrences.

Distribution of specific kinds of smells We investigated in **RQ9.2**, if some specific smells are more prevalent than others. We found that the smells are not equally distributed within the analyzed projects. We also investigated their evolution over the studied releases. Our results show that the studied smells either persist or even mostly increase in number from one release to another. We observed some cases in which there was a decrease from one release to the other, and where smells occurrences were intentionally removed (*Rocksdb*, *Conscrypt*) by refactoring. Those systems are emerging respectively from Facebook and Google. In *Realm*, we also observed the awareness of developers about the bad practice of not removing local references (commit message: `"DeleteLocalRef when the ref is created in loop`

(#3366) Add wrapper class for JNI local reference to delete the local ref after using it"). This could explain the decrease of smells occurrences in some situations. However, since no automatic tool is available, it could be really hard to identify all the occurrences, especially since such systems usually include different teams, which could explain the increase and decrease of multi-language design smells occurrences.

Our results show that *Unused Parameters* is one of the most frequent smells in the analyzed projects. This could be explained by the nature of the smell. This smell is defined when an unnecessary variable is passed as a parameter from one language to another. Since multi-language systems are emerging from the concept of combining heterogeneous components and they generally involve different developers who might not be part of the same team, it could be a challenging task for a developer working only on a sub-part of a project to clearly determine whether that specific parameter is used by other components or not. Thus, developers will probably tend to opt for keeping such parameters for safety concerns. The same goes for *Too Much Scattering* and *Unused Method Declaration*, these smells are defined respectively by occurrences in the code of native methods declarations that are no longer used, and separate and spread multi-language participants without considering the concerns. The number of these smells seems to increase over the releases as shown in Fig. 9.4. Under time pressure the developers might not take the risk to remove unused code, especially since in the case of JNI systems, such code could be used in other components. Similarly, the high distribution and increase of *Too Much Scattering* could be explained in situations where several developers are involved in the same projects, bugs related to simultaneous files changes may occur. When features are mixed together, a change to the behavior of one may cause a bug in another feature. Thus, developers might try to avoid these breakages by introducing scattered participants. Similarly, the design smell *Not Securing Libraries* is prevalent in the analyzed systems. We believe that developers should pay more attention to this smell. Malicious code may easily access such libraries. Occurrences of this smell can introduce vulnerabilities into the system, especially JNI systems that have been reported by previous studies to be prone to vulnerabilities [11, 12]. Several problems may occur due to the lack of security checking. An unauthorized code may access and load the libraries without permission. This may have an adverse impact especially in industrial projects that are usually developed for sale or are available for online use, or other safety-critical systems.

9.2.13 Smells and Faults

Relation Between Smells and Faults In **RQ9.3**, we analyzed the relation between smells and fault-proneness. We used Fisher's exact test and the odds ratios to check whether

the proportion of buggy files varies between two samples (with and without design smells). From our results, we found that in general odds ratios are higher than one. This confirms previous insights from mono-language studies in which researchers claimed that design smells could increase the risk of faults [23, 143]. We cannot claim causation as we do not know whether such faults could have been caused by other factors. Although, our results suggest that files with JNI systems are more likely to be associated with faults than files without. In *Zstd-jni*, we found higher ORs than those of other systems from *13.9285* to *37.7142*; this could be explained by the nature of smells involved in this system as reported in Table 9.5. Some types of smells could be more related to bugs than other types. Out of all the 98 releases analyzed, we found eight releases with ORs less than one, however, none of them was with a significant p-value. In *Java-smt* and *Javacpp*, p-values are not statistically significant (higher than 0.05) in most releases.

From studying bug-fix commit messages, we observed that the impact is also smell-dependent. Occurrences of some types of smells seem more related to bugs than others, which motivates us to perform the **RQ9.4**. Some occurrences of smells related to bugs have been refactored from one release to the next one. In many cases, we find a description in the commit message indicating refactoring for removing specific smells that caused the bugs (commit message: *e.g.,* `"There were a bunch of exceptions that are being thrown from JNI methods that aren't currently declared", "cleaning up JNI exceptions (#252)", "removed a few unused JNI methods"`). Mono-language smells have been widely studied in the literature and were reported to negatively impact systems by making classes more change-prone and fault-prone.

Multi-language systems could introduce additional challenges compared to mono-language systems. Those challenges are mainly related to the incompatibilities of programming languages and the heterogeneity of components. Thus, the design smells occurring on those systems are expected to increase the challenges related to the maintenance of these systems. Even if some smells *e.g., Unused Method Declaration*, *Unused Method Implementation* could not be directly related to bugs, they seem to increase the maintenance efforts because some of them are intentionally removed by developers. Thus, we believe that developers should be cautious about files with design smells, because they are more likely to be subject to faults and thus may incur additional maintenance efforts. Developers should also pay attention to avoid introducing occurrences of such design smells when dealing with JNI systems.

Relation between Specific Smells and Faults Results from **RQ9.4** show that some smells seem more related to faults than others: *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding Libraries*, and *Not Handling Exceptions*. The smell types

Memory Management Mismatch and *Not Securing Libraries* are also found to be related to bugs. We believe that files containing these smells should be considered in priority for testing and-or refactoring. The smell *Not Handling Exceptions* was previously reported as related to bugs [12, 82]. In fact, we discussed a bug related to this smell early in Chapter 1. A bug related to this smell was reported in *Conscrypt*, developers were not checking for Java exceptions after all JNI calls that might throw them. The management of exceptions is not automatically ensured in all the programming languages. Incompatibility between the programming languages may lead to bugs and challenges related to the conversion, management of memory, and other mismatches between programming languages. In JNI projects, developers should explicitly implement the exception handling flow. Similarly, bugs in files containing the smells *Unused Parameters* and *Too Much Clustering* could be explained as the impacts of the noises that these two smells could introduce. Indeed, unused code or huge files with JNI code could impact code maintainability and the comprehension of JNI systems, which may lead to the introduction of bugs. From our detection approach, we identified files containing more than 200 method declarations that are not necessarily related in terms of responsibilities and that do not follow the principle of separating the concerns. We believe that faults could be easily introduced in such files, especially when dealing with JNI code; a developer might not be an expert on all the languages used and the inter-language interfaces. Developers should be concerned about the types of smells that are more likely to introduce bugs. The code containing these smells should be prioritized for testing and refactoring.

9.2.14 Risky Activities

From **RQ9.5**, we found that activities related to data conversion, memory management, restructuring the code, API usage, and exception management are among the activities that could increase the risk of bugs when performed on smelly files. This is not surprising as several articles and developers' blogs discussed bugs related to the management of the memory in JNI systems [11,12]. Some of these activities are directly related to design smells discussed in this thesis, *e.g., Memory Management Mismatch*, *Local References Abuse*. Developers who do not follow good practices to avoid such design smells could perform activities that could increase the risk of future bugs in those files. In the context of JNI systems, it is the developers' responsibility to take care of the management of memory because of the incompatibility between Java and C/C++. The same goes for data conversion when using JNI. We should consider specific rules to convert and access data between Java and C/C++. Primitive types could be easy to convert from Java to C/C++. However, reference types are more complex and require additional knowledge on what kind of methods to use to apply a proper conversion. Several studies also discussed issues related to exceptions in JNI context. Unlike Java, C/C++ does

not support the automatic handling of exceptions. Developers could introduce bugs if they do not have enough knowledge about how to implement the exception handling flow in JNI context. Such incompatibility between programming languages could introduce bugs and other maintenance challenges including checking exceptions, buffer overflows, and memory leaks [63]. Following formal guidelines and being aware of the practices to follow could help to improve the quality of those systems [11, 12, 34, 49]. We also noticed that in some systems, developers started paying more attention to this smell to avoid bugs related to the management of exceptions *Conscrypt*: `"This works towards issue #258. So the exception can be routed out properly, this moves the SSL_get0_peer_certificates call to after doHandshake completes in ConscryptFileDescriptorSocket"`. Another example from *Realm*, developers started paying more attention to the smell *Local References Abuse* `"DeleteLocalRef when the ref is created in a loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it"`. We believe that further investigations should be performed to better understand the reasons for bug introduction in the presence of this smell.

9.2.15 Threats To Validity

In this section, we shed light on some potential threats to the validity of our methodology and findings following the guidelines for empirical studies [118].

Threats to Construct Validity These threats concern the relation between the theory and the observation. In this study, these threats are mainly due to measurement errors. Most of the studied projects rely on Github issues to report bugs. Therefore, we identified fault-fixing commits by mining the Github commit logs using a set of keywords extracted from the literature [17, 132, 144]. We used a set of keywords similar to those previously used in studies focusing on bug prediction. However, this technique may not capture all the commits related to fault-fixing if the commit messages were not representative enough of the developer's intention or were not containing any of those keywords. Nevertheless, this methodology was successfully used in multiple previous empirical studies [24, 31, 138, 144]. Moreover, in [145], the authors report that this technique can achieve a precision of 87.3% and a recall of 78.2%. Another threat to construct validity is related to the accuracy of the SZZ heuristic used to identify fault-inducing commits. Although this heuristic does not achieve a 100% accuracy, it has been successfully employed and reported to achieve good results in multiple empirical studies from the literature [146–148]. We also did a manual validation of the bug inducing commits as described in Section 9.2.5 by inspecting the changes of a small sample of bug inducing commits.

When analyzing the smelliness of files that experienced bugs, we considered the whole file as participating in the design smell. Hence, the smell present in the file could be in different code lines than the bug. There is a similar threat in our analysis of the activities introducing bugs. We rely on commit messages provided by developers to identify the activities. We are aware that in some cases, developers might not have provided all the details of the activities performed or might have used some abbreviations. However, we mitigated this threat by combining both manual and automatic approaches to capture the possible activities that were performed. We are aware that the retrieved topics may not be 100% accurate. However, we followed the coding methodology applied in previous studies [149,150] and two of the research team manually validated a subset of the commit messages. Through the manual analysis, we found that some commit messages describe more than one activity in the same commit (*e.g.,* Commit extracted from *Zstd-jni*: `"Align the JNI names with the new streaming API, Move to the new streaming API, Use the ZBUFF based streaming compression"`) while they assigned by the automatic approach to a single category. Although, the category to which they are assigned is based on the frequencies of the related keywords. We mitigated this threat by performing a manual validation over 500 commit messages. We also investigated some examples of activities at the source code level in the smelly code as described in Section 9.2.10. The list of the activities may not be exhaustive and do not present a 100% recall and precision. However, in this study we are reporting our observation on the activities that once performed in smelly files could introduce bugs without any empirical comparison of the risk introduced by each activity.

Threats to Internal Validity We do not claim causation and only relate the presence of multi-language design smells with the occurrences of faults. We report our observations based on empirical results and explain these observations with manually analyzed examples from the studied systems to better contextualize our findings. We are aware that smells can depend on each other and we select the subset of non-correlated smells while building the logistic regression models. However, the variations in the distribution of smells, and some smells being very infrequent can have negative impacts on the regression models. As our model for each system considers all releases of a particular system than individual releases separately, it helps compensate for the infrequent classes by boosting the per-class data size.

Threats to External Validity These threats concern the possibility to generalize our results. We studied nine JNI open source projects with different sizes and domains of application. We focused on the combination of Java and C/C++ programming languages. Nevertheless, further validation of a larger number of systems with other sets of languages

would give more opportunities to generalize the results. We studied a particular yet representative subset of multi-language design smells. Future works should consider analyzing other sets of design smells.

Threats to Conclusion Validity These threats are related to the relationship between the treatment and the outcome. We were careful to take into account the assumptions of each statistical test. We mainly used non-parametric tests that do not require any assumption about the data set distribution.

Threats to Reliability Validity We mitigate the threats by providing all the details needed to replicate our study in section 9.2. We analyzed open source projects hosted in GitHub. We provide an online access to all the data and scripts used to conduct this study[12].

9.3 Chapter Summary

In this chapter, we investigated the prevalence and impacts of 15 design smells on fault-proneness. We studied the design smells presented in Chapter 7 using the approach introduced in Chapter 8. We showed that the design smells are prevalent in the studied projects and persist across the releases. Some types of smells are more prevalent than others. Our results suggest that files with design smells are more likely to be subject of bugs than files without those smells. We also report that some specific smells, are more likely to be of a concern than others, *i.e., Unused Parameters*, *Too Much Scattering*, *Too Much Clustering*, *Hard Coding Libraries*, and *Not Handling Exceptions*. These smells seem more related to faults, thus we suggest that practitioners consider them in priority for testing and-or refactoring. This empirical study supports, within the limits of its threats to validity, the conjecture that multi-language design smells are prevalent in the studied projects and that similar to mono-language smells, design smells may have a negative impact on software reliability. From analyzing fault-inducing commits we found that data conversion, memory management, code restructuring, API usage, and exception management activities could increase the risk of bug introduction when performed on smelly files. We believe that the results of this study could help not only researchers but also practitioners involved in building software systems using more than one programming language.

[12] `https://github.com/ResearchML/TOSEM_MLS_DesignSmells_Fault`

CHAPTER 10 MULTI-LANGUAGE DESIGN SMELLS AND FAULT-PRONENESS: A SURVIVAL ANALYSIS

10.1 Chapter Overview

In the previous chapter, we performed an empirical investigation of the impact of the design smells presented in Chapter 7 on software fault-proneness [129]. From our analysis in Chapter 9, we found that the risk of bugs is higher in files with multi-language design smells than files without occurrences of multi-language smells. In that study, we investigated the correlation between smells and bugs without considering the time to bug occurrence. Thus, to complement our results presented in the previous chapter, in this chapter, we perform an empirical study investigating the time to bug occurrence in files with and without multi-language design smells using survival analysis. The knowledge about the time to bug occurrence is useful to understand the importance of multi-language design smells and their impacts on software quality. Moreover, having the timeline information could help prioritizing the testing activities by knowing which smells should be considered in priority. Earlier studies investigating the lifespan and impact of design smells on software fault-proneness in mono-language systems also used survival analysis [24,53,107,151]. The objective here is to investigate whether the files with multi-language design smells have lower survival probabilities from bug occurrence compared to files without those smells, to assess the impacts of multi-language smells on fault-proneness. To gain a deep insight into these impacts, we combine quantitative and qualitative analysis.

10.2 Study Design

We present in this section our methodology to perform this study. Figure 10.1 provides an overview of the methodology.

Part of the content of this chapter is submitted for publication as: Mouna Abidi, Md Saidur Rahman, Moses Openja, Foutse Khomh. "Design Smells in Multi-language Systems and Bug-proneness: A Survival Analysis", *in Transactions on Software Engineering and Methodology (TOSEM), 2021, ACM.*

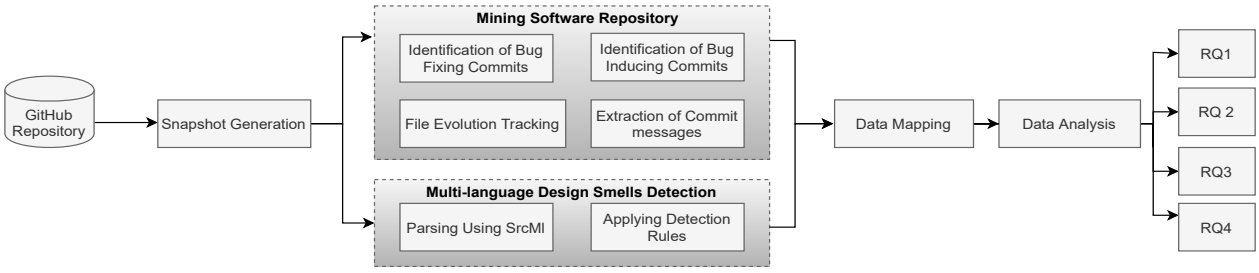


Figure 10.1 Schematic Diagram of the Study

10.2.1 Setting Objectives of the Study

Our goal in this study is to investigate the relation between the occurrence of multi-language design smells and the occurrence of bugs overtime. The quality focus is the survival of multi-language files before the occurrence of a bug and consequently the maintenance efforts due to the presence of multi-language design smells. The perspective is that of researchers, interested in improving the quality of multi-language systems, and to mitigate the impacts of multi-language design smells on software fault-proneness. Practitioners will also benefit from the result of this study, since it will allow them to identify the types of multi-language design smells that are more likely to experience bug fixes. They will also get insights about the types of bugs occurring in these smells. This work could also be of interest to testers who need to prioritize their testing activities and would benefit from knowing which files should be tested in priority. Finally, they can be of interest to quality assurance teams or managers who could use the results of our study to assess the fault-proneness of in-house multi-language projects. This study considers 15 types of multi-language design smells presented in Chapter 7. We detected and analyzed smells from 270 releases of eight open-source multi-language projects. We defined the following research questions to achieve our research objectives:

- **RQ10.1: Is the risk of bugs higher in files with multi-language smells in comparison with those without smells?** Several existing studies investigated the impact of design smells on fault-proneness but primarily in mono-language systems. Thus, through this question, we investigate how long do smelly files survive before a bug occurrence and whether smelly files survive shorter or longer than files without multi-language smells. We define the following null hypothesis: H_0^1*: There is no difference between the probability of a bug occurrence in files with and without smells.*

- **RQ10.2: Is the risk of bugs equal from one multi-language design smell type to the other?** During maintenance activities, developers are interested in identifying parts of the code that should be tested and–or refactored in priority. Hence, we

are interested in identifying the type of multi-language design smells that have higher negative impacts on multi-language systems, *e.g.,* smells that make JNI systems more bug-prone. In particular, we investigate whether some specific types of multi-language smells have shorter or longer survival time from bug occurrence in comparison to other types of smells. We define the following null hypothesis: H_0^2*: There is no difference between the probability of a bug occurrence in files containing one specific type of multi-language design smells and in files with other type(s) of smell(s).*

- **RQ10.3: What are the categories of bugs that exist in multi-language smelly files?** To better capture the impacts of multi-language smells on software fault-proneness, we believe that it is important to investigate the types and characteristics of bugs that exist in multi-language smelly files. Thus, through this research question, we aim to derive insights beyond our quantitative findings through qualitative analysis and provide a categorization and characterization of bugs related to files with multi-language smells.

- **RQ10.4: What are the dominant categories of bugs related to each type of multi-language smell?** As each type of smell points to a specific type of design deficiency, some specific types of design smells could lead to different types of bugs. Consequently, the impacts of the smells on the software quality are likely to vary. Hence, we aim through this question to study the categories of bugs related to each specific smell type.

10.2.2 Data Collection and Data Extraction

Data Collection

Our empirical study is based on eight open-source multi-language projects. We selected these projects because they are well-maintained and highly active projects on GitHub. Another criteria for the selection was 'diversity', *i.e.,* those systems are from diverse application domains, of different sizes, with varying distributions of multi-language code and differ in lengths of development periods. Table 10.1 summarizes the characteristics of the selected systems.

Snapshot Generation

We used Python scripts to mine repositories on GitHub. For each system, we clone the repository from GitHub and extract commit logs. We identify snapshots at every 90 days interval based on the commit logs. We then extract the selected versions to create snapshots of the code for analysis. We considered snapshots at each 90 days interval from the first commit to the last commit. The number of snapshots varies across the systems. In cases where a snapshot is not available exactly at 90 days interval, we consider the next available commit after 90 days. We selected 90 days to take quarterly snapshots, which is a widely used milestone in product road-map timeline [152]. We selected a total of 270 snapshots from the eight subject systems (shown in Table 10.1).

Table 10.1 Overview of the Studied Systems

Projects	**Domain**	**#Snapshot**	**LOC**	**Java**	**C/C++**
Rocksdb	Facebook Database	36	487853	11%	83.1%
Frostwire	File and Media Sharing	18	403 106	71.4%	19%
Realm	Mobile Database	29	171705	82%	8.1%
Conscrypt	Cryptography (Google)	32	91765	85.3%	14%
Pljava	Database	35	71910	67%	29.7%
Javacpp	Compiler	30	28713	98%	0.6%
JNA	Native Shared Library	32	590 208	70.2%	15.4%
OpenDDS	Adaptive Communication	58	2803495	5%	16%

Identification of Bug-Fixing and Bug-Inducing Commits

To identify bug-fixing commits, we used a set of error related keywords (*e.g., fix, fixed, fixes, bug, error, except, issue, fail, failure, crash*) as described in Section 9.2.5. We used a Python script that fetches commits containing at least one keyword from the set of keywords in the commit messages as bug-fixing commits. Similar to the study presented in Chapter 9, we used PyDriller to capture the bug-inducing commits for each of the bug-fixing commits [107, 129, 153–156]. We use these bug-inducing commit dates to calculate the distance (in hours) from the file creation dates to determine the survival time for our survival analysis. We decided to compute the distance in hours to reduce possible noise (*e.g.,* due to day-level rounding of timestamp) in the time data of our dataset.

File History Tracking

Changes are part of the project development cycle. During the life cycle of a project, files could be added, removed, modified, renamed or even relocated. Thus, tracking the file genealogy is necessary to ensure the reliability of the results that are based on the evolution

history of files. We use the `git diff` command to compare changes between two consecutive snapshots. The command returns the list of files that have been either added, modified, renamed, or deleted between two given commits. The command provides an estimation on how likely a specific file was renamed. Similar to previous work, we relied on a similarity threshold of 70% to identify renamed files [151]. Our file tracking assigns unique IDs for individual files and ensures that different versions of the same file take the same ID despite renaming or relocation of the files.

Design Smells Detection

We relied on the multi-language smell detection approach we described in Chapter 8. As described earlier, the detection approach supports the 15 types of multi-language design smells described in Chapter 7.

Mapping Bug-Inducing Commits with Design Smells

Since our goal is to investigate the impact of multi-language design smells on software fault-proneness, the occurrences of the studied smells should exist in the files before the bugs occur. This was ensured using the bug-inducing commits retrieved by PyDriller. For a specific snapshot S_t at time t corresponding to a commit in the repository, we detect the smells occurrences. Then, we identify the bug-inducing commits between S_t and $S_{t+\alpha}$ that contain the smelly files from version S_t. We considered an α of 90 days (or the duration in days to next available commit after 90 days) defining the time intervals between two consecutive snapshots. We also collected the bug-inducing commits for non-smelly files using the same methodology since our objective is to compare the survival times to bug occurrence in files with smells and that of files without those smells using survival analysis.

10.2.3 Data Analysis

In the following subsections, we present the analysis we performed to answer our research questions.

Survival Analysis

Survival analysis are used to model the expected duration of time until the occurrence of one or more event(s) of interest. One of the most popular models for survival analysis is the Cox Hazard model, *i.e.,* Cox Proportional Hazards model. Cox Hazard model analyze how long specific subjects can survive before a well-defined event occurs. In this study, for each file, we

apply the Cox model to compute the risk of bug occurrence over time (in hours), considering a number of independent covariates. We use the Cox hazard model as it allows subjects (files) to remain in the model for the whole observation period even if they do not observe the event (bug occurrence). The subjects can also be grouped based on the covariates (*e.g.,* smelly and non-smelly) and the model is also able to accommodate the changes in characteristics of the subjects over time.

Topic Modeling

To categorize the bugs existing in multi-language smelly files, we merged all the bug-fixing commit messages related to smelly files for all the systems in one file. We use this file as the corpus for the topic modelling to extract the topic of bugs. We applied topic modeling as described in Section 10.2.3. To discover the optimal number of topics K, we experimented with different values of K ranging from 5 to 50, increasing K by 5 at each time. From our analysis, the LDA generates 20 topics. Each of those topics contains the list of commit messages used to build that topic along with their probability score. We then perform manual analysis to assign meaningful names to each topic using the keywords and randomly selected top 20 to 30 documents for each topic as performed in previous studies [157]. The topic files were shared between the two members of the research team. The topics were assigned after reaching an agreement. We also group the bug topics into general categories of bugs that we report in Section 10.3.

10.3 Study Results

In this section we report our findings and answer each of the four research questions.

Table 10.2 Bug Hazard Ratios for Each Project

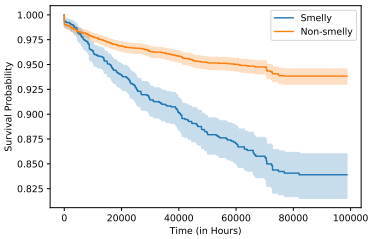
Projects	**exp(coef)**	**p-value (CHM)**	**p-value (PHA)**
Rocksdb	1.64	6.162e-26	1.258e-05
Frostwire	3.123	1.749e-52	0.641
Realm	2.747	7.487e-37	9.112e-05
Conscrypt	2.598	3.218e-23	0.0001
Pljava	1.805	6.425e-05	0.002
Javacpp	2.378	3.003e-08	0.164
JNA	5.033	9.526e-32	1.254e-14
OpenDDS	0.229	1.468e-09	0.992

CHM: Cox Hazard Model, **PHA:** Proportional Hazards Assumption

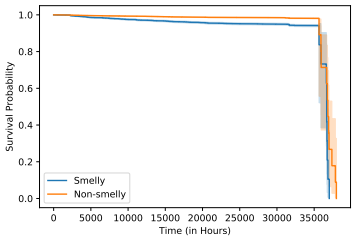
RQ10.1: Is the risk of bugs higher in files with multi-language smells in comparison with those without smells?

Approach To assess the impacts of multi-language design smells on software fault-proneness, we performed survival analysis and compared the time until the occurrence of bug in smelly and non-smelly files. First, we detect multi-language design smells for each snapshot by using the approach proposed in Chapter 8. We used PyDriller to capture the information of bugs at the file level. For each system and for each snapshot, we compute the following metrics for each file: **Time** represents the survival time as the number of hours from file creation to first occurrence of bugs. For files without bugs we assign the survival time as the difference between the file creation time to the end of the analysis period for that system. **Smelly** illustrates the covariate of interest. This variable takes 1 if the file contains at least one type of design smell in a specific snapshot and 0 otherwise. **Inducing_Flag** reflects our event of interest. It takes 1 if a specific file i is reported by PyDriller as containing a bug-inducing commit in that specific snapshot. We divide our dataset into two groups, One group for files with multi-language smells and the second group for files without any of the studied multi-language smells. We create Cox model for each of these groups and used R functions (*i.e., Surv* and *coxph*) to analyze the Cox model. Since the covariate of interest for this study (*i.e.,* the **Smelly** metric) is a constant function that presents 1 or 0, thus, for this analysis we can demonstrate a linear relationship with the event of interest without using a link function, similarly to what has been done in previous studies investigating the impact of mono-language design smells on JavaScript projects [24, 151].

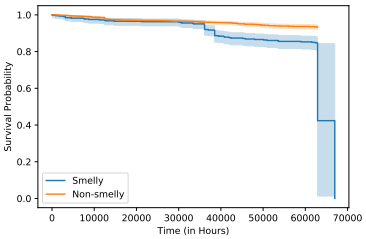
Findings Figure 10.2 provides the survival curves for native multi-language files with and without smells for all eight studied systems. Results presented in Fig. 10.2 show that, in the majority of the studied systems, files with multi-language design smells experience bugs faster than files without those smells. The X-axis presents the survival time in hours while the Y-axis presents the survival probability of a file until the occurrence of a bug. Thus, a low survival rate is expressed by a low value on the Y-axis. We can clearly see from the survival curves in Fig. 10.2 that files with multi-language design smells in most of the selected systems experience bugs faster in comparison with non-smelly files. For all our subject systems, we compute the hazard rates between files with and without design smells. We also performed log-rank test to statistically compare the survival distribution of files with and without smells. Table 10.2 presents the bug hazard ratios for each system. All the systems except OpenDDS present hazard ratios (*exp(coef)*) greater than 1. This provides an evidence that files with multi-language design smells are at higher risk of bugs compared to files without multi-language smells.



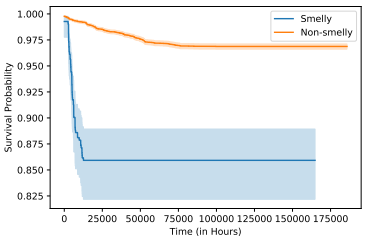
(a) Conscrypt



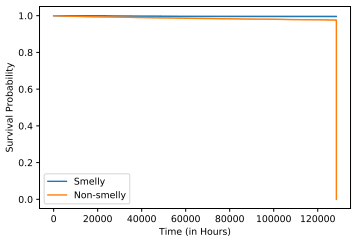
(b) Frostwire



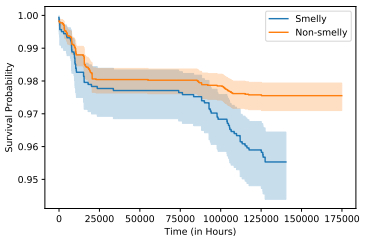
(c) Javacpp



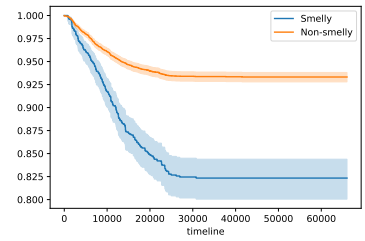
(d) Jna



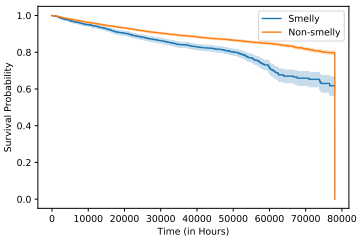
(e) OpenDDS



(f) Pljava



(g) Realm-java



(h) Rocksdb

Figure 10.2 Survival Curves for Bug-occurrences in Files with (Smelly) and without (Non-smelly) Multi-language Smells

We believe that the results of hazard ratios computed for OpenDDS could be related to the type of design smells existing in this system, which motivated us to investigate in **RQ10.2** the survival until a bug occurrence for each type of smell. Our results for **RQ10.1** show that files without multi-language design smells have hazard rates lower than files with multi-language smells in 87.5% cases *i.e.,* in 7 out of 8 systems. From the log-rank test, we obtained for all the subject systems the p-values (Cox hazard model) less than 0.05. Thus, we reject H_0^1. Therefore, we conclude that the risk of bug occurrence is higher (bugs occur faster) in files with multi-language design smells compared to files without those smells.

Summary of findings (RQ10.1): Files with multi-language design smells experience bugs faster than files without those smells. Files without multi-language smells have hazard rates lower than files with multi-language smells in 87.5% cases and this difference is statistically significant.

RQ10.2: Is the risk of bugs equal from one multi-language design smell type to the other?

Approach Given the observed impacts of the multi-language smells on fault-proneness in RQ10.1, it is important to investigate the impact of each type of multi-language design smells. We perform survival analysis for each type of smells, comparing the time until occurrence of a bug in files containing that specific smell and files without that smell. We follow the same approach as in **RQ10.1** and compute the metrics **Time** and **Inducing_Flag**. We also compute the metric $\textbf{Smelly}_i$ which takes the value 1 if the file contains the smell type i in the specific snapshot r and 0 if it does not contain any smell of that type. Because the following metrics are known to be related to fault-proneness [24,137,138], we add the file size (LOC), and the number of previous occurrence of bugs to our model, to control their effect. Here, (i) **LOC** refers to the number of lines of code in the file at that specific snapshot; (ii) **N.Previous-Bugs** is the number of bug-fixing related to that file before the snapshot r.

Findings We present the survival curves for smelly and non-smelly files for individual smell types in Fig. 10.3 (*Conscrypt*), Fig. 10.4 (*Frostwire*), Fig. 10.5 (*Javacpp*), Fig. 10.6 (*JNA*), Fig. 10.7 (*OpenDDS*), Fig. 10.8 (*Pljava*), Fig. 10.9 (*Realm*), and Fig. 10.10 (*Rocksdb*). These survival curves in general show that for most of the smell types, files with given type of smells tend to have lower survival probability compared to files without those smells. This is an indication that files with multi-language smells are at higher risk of bugs than files without smell. In Table 10.3 and Table 10.4, we present the bug hazard ratios for

Table 10.3 Hazard Ratios for Each Type of Multi-language Design Smells (higher exp(coef) values means higher hazard rates) 1/2

Projects	Covariate	exp(coef)	Rank	p-value (CHM)	p-value (PHA)
Rocksdb	LOC	1.64	–	6.162e-26	1.258e-05
	EIC	0.610	5	0.019	0.144
	TMC	0.573	6	0.0012	0.008
	TMS	0.22	–	2.246	0.194
	UMD	0.876	–	0.598	0.130
	UP	2.677	4	2.451e-87	3.741e-06
	ASRV	3.186	3	3.116e-08	0.177e-06
	NHE	3.186	3	3.116e-08	0.177e-06
	NSL	1.064	–	0.846	0.098
	HCL	2.256	–	0.972	0.999
	NURP	2.062	–	0.05	0.039
	MMM	3.279	2	0.0003	0.087
	LRA	5.10	1	2.677e-07	0.939
Frostwire	EIC	1.065	–	0.95	0.155
	TMC	0.975	–	0.98	0.135
	UMD	2.252e-06	–	0.980	0.997
	UMI	2.279	–	0.41	0.153
	UP	3.163	1	1.206e-53	0.617
	ASRV	6.138e-06	–	0.984	0.997
	NHE	2.25	–	0.985	0.998
	NSL	0.339	–	0.279	0.146
	NURP	0.403	–	0.365	0.144
Realm	PrevBugs	2.746	–	7.487e-37	9.1123e-05
	EIC	5.015	5	2.426e-31	0.0945
	TMC	4.729	6	2.17e-32	0.001
	TMS	2.259	9	0.0008	0.69
	UMD	5.305	4	1.693e-13	0.01
	UMI	5.717	3	1.136e-12	0.043
	UP	1.996	10	2.126e-11	4.169e-05
	ASRV	2.482	8	0.010	0.913
	NHE	1.91	11	0.005	0.40
	NSL	2.922	7	0.009	0.958
	MMM	6.88	2	8.748e-09	0.21
	LRA	6.975	1	0.0001	0.0020
Conscrypt	LOC	2.59	–	3.218e-23	0.0001
	EIC	7.72	6	1.393e-21	0.0009
	TMC	8.025	5	5.368e-29	0.178
	UMD	8.088	4	1.228e-31	0.461
	UP	1.889	9	5.017e-10	0.0013
	ASRV	8.278	3	1.117e-15	0.196
	NHE	8.278	3	1.117e-15	0.196
	NSL	2.345	8	0.025	1.113e-06
	NURP	2.825	7	0.0065	4.411e-06
	MMM	19.60	1	1.924e-27	0.0002
	LRA	19.077	2	1.167e-37	0.0001

Acronyms: NURP: NotUsingRelativePath, **TMS:** ToomuchScattering
TMC: Toomuchclustring, **EILC:** ExcessiveInterlangCommunication
ASR: AssumingSafeReturnValue, **UM:** UnusedMethodDeclaration
NSL: NotSecuringLibraries, **NHE:** NotHandlingExceptions
MMM: MemoryManagementMismatch, **LRA:** LocalReferencesAbuse
CHM: Cox hazard Model, **PHA:** Proportional Hazards Assumption

Table 10.4 Hazard Ratios for Each Type of Multi-language Design Smells (higher exp(coef) values means higher hazard rates) 2/2

Projects	Covariate	exp(coef)	Rank	p-value (CHM)	p-value (PHA)
Pljava	LOC	1.80	–	6.43e-05	0.0027
	EIC	0.565	–	0.569	0.58
	TMC	1.008	–	0.988	0.339
	TMS	0.833	–	0.525	0.101
	UMD	1.217	–	0.404	0.028
	UMI	1.867	–	0.38	0.611
	UP	2.006	3	2.590e-05	0.004785
	NHE	5.666	2	0.0001	0.082
	NSL	1.67e-05	–	0.991	0.91
	NURP	1.669e-05	–	0.991	0.99
	MMM	2.613	–	0.176	0.877
	LRA	26.196	1	5.31e-30	1.857e-05
Javacpp	EIC	1.604	6	0.021	0.089
	TMC	3.443	5	1.699e-05	0.314
	TMS	1.310	–	0.1612	0.164
	UMD	3.875	4	7.218e-11	0.008
	UP	9.53	3	0.001	0.34
	NSL	11.94	2	9.3172e-26	0.003
	HCL	16.35	1	4.397e-13	8.184e-06
JNA	PrevBugs	5.033	–	9.526e-32	1.254e-14
	EIC	33.518	7	1.156e-17	0.139
	TMC	30.929	9	1.830e-19	0.177
	TMS	30.71	10	2.142e-24	0.028
	UMD	235.726	1	5.60e-08	0.100
	UMI	168.518	2	3.295e-07	0.113
	UP	3.911	11	1.196e-18	1.192e-11
	ASRV	161.083	3	4.155e-07	0.111
	NHE	69.77	4	4.153e-21	0.612
	NSL	62.422	5	1.893e-16	0.533
	NURP	62.422	5	1.893e-16	0.533
	MMM	60.852	6	2.919e-16	0.613
	LRA	33.298	8	1.343e-17	0.163
OpenDDS	LOC	0.229	–	1.469e-09	0.992
	TMC	2.251e-06	–	0.968	0.999
	TMS	8.228e-07	–	0.966	0.999
	UMD	2.2498e-06	–	0.966	0.999
	UMI	6.133e-06	–	0.967	0.999
	UP	0.273	1	1.346e-06	0.468
	ASRV	0.732	–	0.59	0.177
	NHE	0.612	–	0.395	0.178
	NSL	2.251	–	0.968	0.999
	NURP	6.139e-06	–	0.978	0.999
	LRA	6.139e-06	–	0.9787	0.999

Acronyms: NURP: NotUsingRelativePath, **TMS:** ToomuchScattering
TMC: Toomuchclustring, **EILC:** ExcessiveInterlangCommunication
ASR: AssumingSafeReturnValue, **UM:** UnusedMethodDeclaration
NSL: NotSecuringLibraries, **NHE:** NotHandlingExceptions
MMM: MemoryManagementMismatch, **LRA:** LocalReferencesAbuse
CHM: Cox hazard Model, **PHA:** Proportional Hazards Assumption

different multi-language design smells existing in the studied systems. The value reported in the column *exp(coef)* shows the increase or decrease in the likelihood of the hazard (bug occurrence) that is expected for each unit increase in the value of the corresponding covariate. When the hazard ratio for a covariate (predictor) is close to or equal to 1, the covariate does not affect survival of the subject. Hazard ratio less than 1 indicates that the covariate is protective (i.e., the covariate contribute to improved survival) and if the hazard ratio is greater than 1, then the covariate contributes to increased risk (or decreased survival). The cases where the p-values are significant (<0.05) are underlined in Table 10.3 and Table 10.4. So, to evaluate the fault-proneness of individual types of multi-language smells from survival analysis perspective, we examine the values of the hazard ratios ($exp(coef)$) and especially the percentages of cases (systems) where the hazard ratios are greater than 1. As we observe in Table 10.3 and Table 10.4, for most of the studied systems (except *Frostwire* and *OpenDDS*) the majority of the smell types have hazard ratios greater than 1, indicating that the files with different multi-language smells are more bug-prone compared to the files without those smells. However, the hazard ratios vary across the types of multi-language smells and across the studied systems.

To have a better view of the comparative fault-proneness of individual types of multi-language smells, we present a summary of the results presented in Table 10.3 and Table 10.4 in Table 10.5. Here, each row in Table 10.5 shows the summary of the survival analysis results for all the eight studied systems for a specific type of smell. For example, for the smell type *Unused Parameters*, the Cox Hazard Models successfully compute the hazard ratios (HR) for all the eight (8) systems we studied, given the constraints on the data requirements and statistical assumptions for the Cox models. Out of these eight (8) systems, for 7 systems we have hazard ratios greater than 1 ($exp(coef) > 1$) while for the remaining one (1) system (*OpenDDS*) the hazard ratio is less than 1 (as in Table 10.3 and Table 10.4). We count the metric SFB as the number of systems where the smelly files with the smell type (*Unused Parameters* in this case) have $exp(coef) > 1$. We also count NSFB as the number of systems where the smelly files with the given smell type have $exp(coef) < 1$. Here, SFB represents the number of systems where the files with the given smell is more bug-prone (at higher risk of bugs) than files without those smells, while NSFB shows the number of systems where file with the given smell type are less bug-prone (at lower risk of bugs) compared to files with given type of smells. We also present the corresponding percentages for the SFB and NSFB, having values 87.50% (7/8) and 12.50% (1/8) for *Unused Parameters*. For smell types (*Excessive Objects* and *Not Caching Objects*) where we could not compute hazard ratios for the studied systems, we report N/A in the Table. We ranked the percentage values (% SBF) to find the top five smell types that are relatively more bug-prone indicated by the

Table 10.5 Summary of the Comparative Fault-proneness of Different Types of Multi-Language Smells

Smell Type	**#System**	**SFB**	**NSFB**	**% SFB**	**% NSFB**
Unused Parameters	8	7	1	87.50%	12.50%
Unused Method Declaration	8	5	3	62.50%	37.50%
Too Much Scattering	6	3	3	50.0%	50.0%
Too Much Clustering	8	5	3	62.50%	37.50%
Unused Method Implementation	5	4	1	80.0%	20.0%
Assuming Safe Return Value	6	4	2	66.67%	33.33%
Excessive Objects	0	N/A	N/A	N/A	N/A
Excessive Inter-language Communication	7	5	2	71.43%	28.57%
Not Handling Exceptions	7	6	1	85.71%	14.29%
Not Caching Objects	0	N/A	N/A	N/A	N/A
Not Securing Libraries	8	6	2	75.0%	25.0%
Hard Coding Libraries	2	2	0	100.0%	0.0%
Not Using Relative Path	6	3	3	50.0%	50.0%
Memory Management Mismatch	5	5	0	100.0%	0.0%
Local References Abuse	6	5	1	83.33%	16.67%

SFB: %Systems where smelly files are more bug-prone than non-smelly files
NSFB: %Systems where files without (specific) smells are more bug-prone than smelly files
#System: No. of Systems where we have hazard ratios for the concerned smell (covariate)
* Underlined percentage values indicate the top-5 bug-prone smell types

underlined percentage values in Table 10.5. Based on the summary, we observe that the smell types *Hard Coding Libraries* (100.0%, 2/2), *Memory Management Mismatch* (100.0%, 5/5), *Unused Parameters* (87.5%, 7/8), *Not Handling Exceptions* (85.71%, 6/7), and *Local References Abuse* (83.33%,5/6) are more bug-prone compared to other types of smells. Our results show that different types of multi-language smells have different impacts on fault-proneness. We therefore reject the null hypothesis H_0^2.

To observe the comparative hazard ratios of individual types of smells in the studied systems, we also assign ranks to the smell types based on the corresponding values for hazard ratios. The smell type with the highest value for hazard ratio is assigned rank 1, the next is assigned rank 2, and so on. We do not include the covariate of controls (*i.e.,* LOC and N.Previous-Bugs) and rank only the smells. For each system, we consider only the smells with statistically significant p-values for the Cox model (as underlined in Table 10.3 and Table 10.4) for the ranking. Thus, the ranking focuses only on the covariates with statistically significant relationships with bug occurrence, the event of interest. From the ranking we observe that

smell types *Local References Abuse*, *Unused Parameters*, *Memory Management Mismatch*, *Not Handling Exceptions*, *Assuming Safe Return Value*, and *Unused Method Declaration* frequently appear in the top 5 smell types posing higher risk of bugs. This also generally agrees with the result based on the percentage of systems with hazard ratios greater than 1 for the individual smell types. We also observe that smell types *Excessive Inter-language Communication*, *Not Securing Libraries*, *Unused Method Implementation*, *Too Much Clustering* and *Not Using Relative Path* sometimes appear in the top 5 smells with higher risk of bugs.

Also, the covariates 'N.Previous-Bugs' and 'LOC' are in some systems significantly related to occurrences of bugs similar to what was reported in previous studies [24, 151]. However, their hazard ratios are less than that of many of the studied multi-language smells. Thus, we believe that monitoring the file size and number of previous bugs may not be enough to effectively track the fault-proneness of multi-language files. Hence, we recommend to consider prioritizing files containing the studied multi-language design smells during maintenance activities for quality assurance.

Summary of findings (RQ10.2): Multi-language design smells are not equally bug-prone. Design smell types *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, and *Unused Method Declaration* are more bug-prone compared to other types of multi-language smells. Thus, these design smell types should be prioritized during maintenance.

RQ10.3: What are the categories of bugs that exist in multi-language smelly files?

Approach To gain better insights into the impacts of multi-language smells on software fault-proneness, we performed a qualitative analysis using both manual analysis and automated topic modeling to extract the categories of bugs related to multi-language smells. Developers often leave important information in commit logs while fixing software bugs. Such information may include an indication of the root cause of the bug and how it affects the software functionality. We collected the bug-fixing commits and applied topic modeling followed by a manual classification to assign meaningful topics to bugs; allowing us to gain insights about the types, characteristics, and causes of bugs related to multi-language smells.

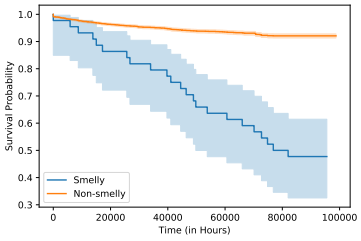
Findings As explained in Section 10.2.3, our topic modeling resulted in 20 topics. We then performed a manual analysis and provided tags for each topic that describe the types of bugs. Each topic refers to a type of bug. We then grouped them into some top-level

Table 10.6 Categories of Bugs and their Distribution in the Dataset

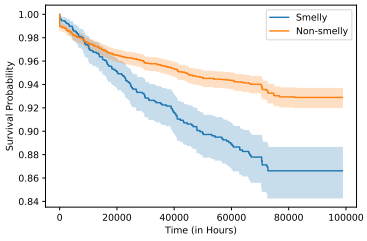
Nc	Bug Topics	%Per	Example of Keywords	Categories
0	Memory Issues	5.65%	memtable, compaction, size, write, flush, datum, set, range, read, trigger.	Memory
1	Performance Issues	3.27%	speed, improve, performance, issues, change, time, execution, work, copy.	Performance
2	Synchronization Issues	4.8%	key, asynchronous, error, synchronized, signature, thrpt_op, mutex_thrpt.	Concurrency
3	Deployment and Environment	5.05%	test_plan, windows, run, fail, make, timeout, 32-bit, deployed, patch.	Platform and Dependencies
4	Support and Features Updates	3.4%	update, change, feature, add, pass, lib, call, load, reference, format.	Libraries and Features Support
5	Threads and multithreading	2.17%	lock, run, Optimistic, thread, error, write, mutex, pass, access, local.	Concurrency
6	Data Schema and Types	7.59%	schema, type, int, object, field, class, default_value, method, pointer,load.	Programming Errors
7	Network Session Handling	3.59%	call, session, nativecrypto, server, cipher_suite, certificate, client, socket.	Communication and Network
8	Distributed Database Storage	6.39%	table, closes_differential, access, key, db, iterator, delete, change.	Database
9	Native Libraries and Platform	4.53%	call, native, failure, external, error, include, load, crash, library, platform.	Libraries and Features Support
10	Communication Protocol	3.59%	file, I/O, user, FIFO, time, input, operation, protocol, revision.	Communication and Network
11	Compiler and Build	4.71%	fail, error, fix, warning, build, make, include, hash, compiles_reviewer.	Programming Errors
12	Data Load and Allocation	3.66%	memory, size, fragments, allocate, create, malloc, cleanup, instance, load.	Memory
13	Breaking Changes	5.83%	support, fix, break, breaking_update, error, add, rename ,library, update.	Programming Errors
14	Missing Dependencies	2.88%	dependencies, libraries, merge, support, missing, JNI, bit, ad, empty.	Libraries and Features Support
15	Network Security	0.79%	synchronization, openssl, java_injecte, nativecrypto, native, sslparameter.	Communication and Network
16	User Interface	1.46%	context_menu, layout, android, cleanup, update_translation, tab, rotation	Programming Errors
17	Programming and Semantics	5.57%	fix, wrong, code, debug, issue, change, cleanup, exception, type, incorrect.	Programming Errors
18	Third Party Libraries	9.8%	android, libraries, issue, external, play, load, media, show, network, music.	Libraries and Features Support
19	Program Compatibility	15.48%	issue, incorrect, warning_native, JNI, Compatibility, checking_exception.	Programming Errors

Table 10.7 Distribution of the Categories of Bugs

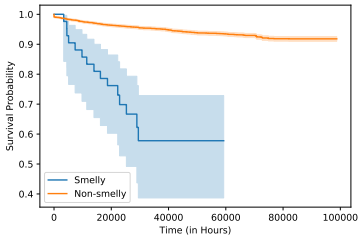
Bug Categories	Percentage
Programming Errors	40.42%
Libraries and Features Support	20.61%
Memory	9.31%
Communication and Network	7.97%
Concurrency	6.98%
DataBase	6.39%
Platform and Dependencies	5.05%
Performance	3.27%



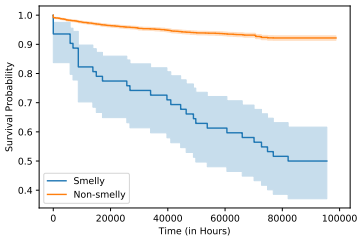
(a) Conscrypt-EIC



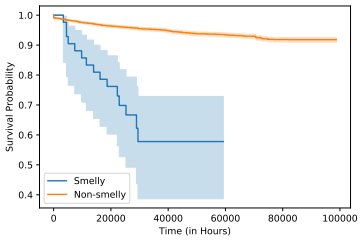
(b) Conscrypt-UP



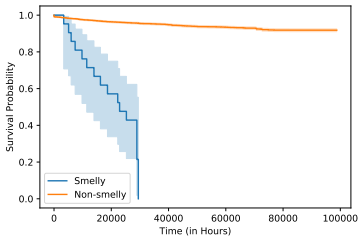
(c) Conscrypt-NHE



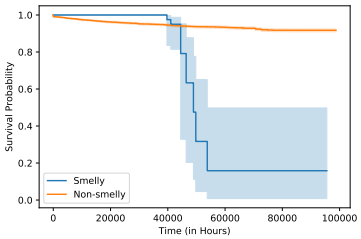
(d) Conscrypt-TMC



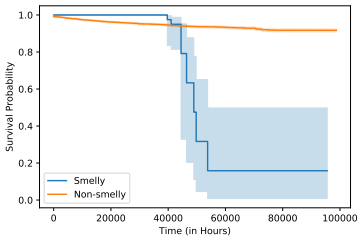
(e) Conscrypt-ASRV



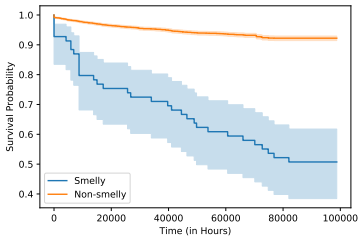
(f) Conscrypt-MMM



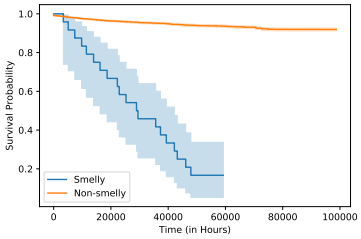
(g) Conscrypt-NSL



(h) Conscrypt-NURP



(i) Conscrypt-UMD



(j) Conscrypt-LRA

Figure 10.3 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Conscrypt

bug categories based on the bug similarity perceived from the bug-fixing commit messages. We extend the bug categorization proposed by Ray *et al.* [17] to include bugs related to multi-language smelly files. Table 10.6 presents the types of bugs extracted from the analysis of commit messages and the corresponding percentage regarding the total number of bugs in all the studied systems. In this table, the column *Bug Topics* provides the type of bugs resulting from the combination of the manual and automatic approach presented in Section 10.2.3, while the column *Categories* illustrates the top-level bug categories that we assigned to group *Bug Topics* based on the similarity in characteristics of the bugs perceived from the bug-fixing commit messages.

Similar to previous work [3, 17], we found that *programming errors* is the largest category of bugs related to multi-language smelly files (40.42%) as shown in Table 10.7. Such proportion is not surprising because this category covers generic programming errors. The highest proportion of bugs in the programming errors category is related to program compatibility (15.48%). As multi-language development involves programming languages with different lexical, semantic, and syntactic rules, developers should be cautious when developing such systems to avoid program incompatibility issues. From analyzing the commit messages, we noticed that such incompatibilities regroup some of the smells discussed in this thesis *e.g., Not Handling Exceptions* and *Memory Management Mismatch* (example of bug-fixing commit message from *Rocksdb*: `"Fixed various memory leaks and Java 8 JNI Compatibility WARNING in native method: JNI call made without checking exceptions when required to from CallObjectMethod"`). Indeed, JNI code requires additional checks and type conversions to correctly define interface between the Java and C/C++ code. Violations of fundamental rules to connect JNI code may lead to bugs.

Libraries and features support (20.61%) is the second largest category of bugs extracted from our dataset. This category comprises the usage of third party libraries and the missing dependencies (example of bug-fixing commit message from *Conscrypt*: `"add missing libraries to JNI lib Conscrypt: fixing Android.mk dependencies Initial empty"`). The integration of third party libraries presents one of the dominant bug topics included in the *external libraries* category (9.8%). Reuse of existing components and libraries are among the most important benefits of multi-language development. Programming languages do not necessarily use the same syntax and semantics. Therefore, it is the developers' responsibility to deal with the different programming languages' calling conventions to avoid diverse issues that can affect software quality. Our results also point to one of the most discussed JNI issues in the literature, the *memory bugs* (9.31%) [11, 82]. Such type of bugs are obvious as programming languages with unmanaged memory type allocations are known for bugs related to memory management [3]. In JNI development, when convert-

ing types or passing an object from Java to C code, the memory is allocated and should be freed after usage. Thus, forgetting to release the memory could introduce memory leaks and bugs. The following commit message related to this category was extracted from *Rocksdb*: `"fix memory leak in two_level_iterator Summary: this PR fixes a few failed contbuild: 1. ASAN memory leak in Block::NewIterator"`.

Our results also report bugs related to *communication and network* with a distribution of 7.97% among our dataset. This result could be explained by the integration of JNI development in network operations in some of the studied projects (*e.g., Conscrypt*, *Frostwire*, and *OpenDDS*). [1] Our results also report the bug category *platform and dependencies* covering 5.05% of bugs. Indeed, using JNI the project loses the platform portability offered by Java code. Therefore, the code should be adapted and compiled to correctly run on different platforms.[2] (`"Fix Windows environment issues, mac and our dev server has totally different definition of uint64_t, therefore fixing the warning in mac has actually made code in linux uncompileable"`) is an example extracted from *Conscrypt* related to fixing bugs related to the category *platform and dependencies*.

Summary of findings (RQ10.3): Programming errors, libraries and features support, and memory issues are the most dominant categories of bugs occurring in files with multi-language design smells.

RQ10.4: What are the dominant categories of bugs related to each type of multi-language smell?

Approach To complement the results of the previous research question (RQ10.3) and to better understand the relationships between the types of bugs and the studied multi-language design smells, we extract the types of smells related to each bug category. In particular, here we study the distribution of the different types of multi-language design smells in the buggy files. We reused the categories and bug types resulting from the previous research question **(RQ10.3)**. Then, we investigated the occurrences and distribution of individual types of multi-language design smells that exist in each buggy file. Note that the categorization of smells and bugs was done by investigating the distribution of smells existing in each file that was reported to be buggy and that was assigned to a category. Thus, design smells could be part of more than one category.

[1] `https://www.ibm.com/developerworks/library/j-transparentaccel/index.html`

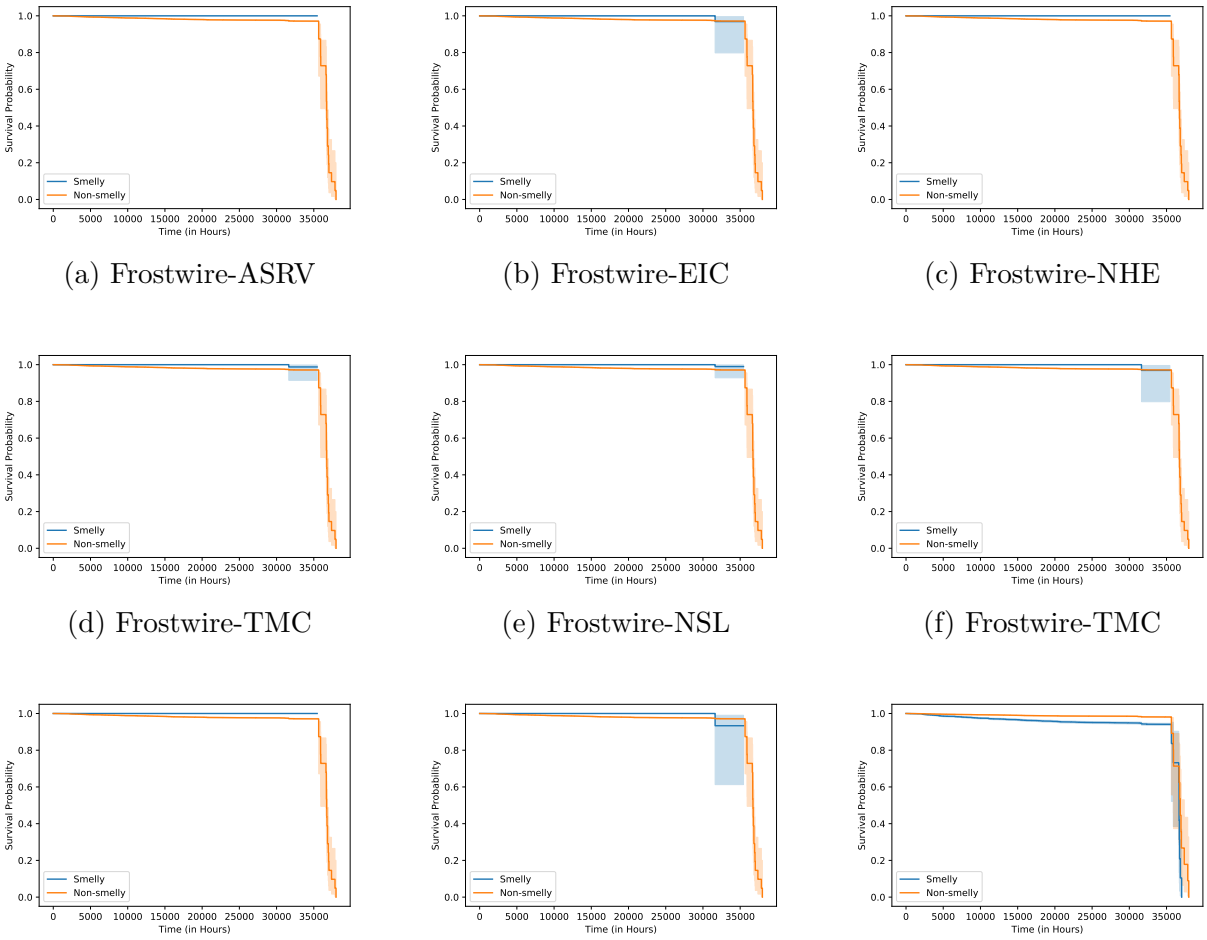
[2] `https://medium.com/swlh/introduction-to-java-native-interface-establishing-a-bridge-between-java-and-c-c-1cc16d95426a`

Listing 10.1 Example of Bug-fixing Commit in Smelly Method

```
/* C */
 void Java_org_rocksdb_Options_setMaxBytesForLevelMultiplierAdditional(
    jintArray jmax_bytes_for_level_multiplier_additional) {
  jsize len = env->GetArrayLength(jmax_bytes_for_level_multiplier_additional);
  jint *additionals =
  -  env->GetIntArrayElements(jmax_bytes_for_level_multiplier_additional, 0);
  +  env->GetIntArrayElements(jmax_bytes_for_level_multiplier_additional,
      nullptr);
  + if(additionals == nullptr) {
  +   // exception thrown: OutOfMemoryError
  +  return;
  +}
  +
  auto* opt = reinterpret_cast<rocksdb::Options*>(jhandle);
  opt->max_bytes_for_level_multiplier_additional.clear();
  for (jsize i = 0; i < len; i++) {
    opt->max_bytes_for_level_multiplier_additional.push_back
    (static_cast<int32_t>(additionals[i]));
  }
  +    (env->ReleaseIntArrayElements(jmax_bytes_for_level_multiplier_additional,
  +      additionals, JNI_ABORT);
```

Findings Table 10.8 shows the distribution of the studied different types of multi-language smells among the categories of bugs. From these results, we observe that the distribution of multi-language design smells differs from one category to the other. The design smells *Unused Method Declaration*, *Excessive Inter-language Communication*, *Unused Parameters*, *Not Handling Exceptions* are the most dominant types of smells among the bug categories.

The design smell *Excessive Inter-language Communication* occurs with an average proportion of 18.82% of all the categories of bugs. This could be explained by the nature of the smell. Indeed, having an excessive communication between code written in different components could increase code maintenance activities and thus the risk of bugs. This smell seems to be highly related to performance issues (43.5%). The design smell *Excessive Inter-language Communication* is also frequently occurring in the category of libraries and features support (28.02%), closely followed by the category of memory bugs (23.61%). This design smell is also frequently occurring in the category of bugs related to general programming errors (19.4%) and errors related to the platform and dependencies (19.05%). Similarly, the design smells *Unused Parameters* and *Unused Method Declaration* occur respectively with average proportion of 18.46% and 16.59% in all the categories of bugs. The design smells *Not Handling Exceptions*, *Too Much Scattering*, and *Too Much Clustering* also occur frequently with average



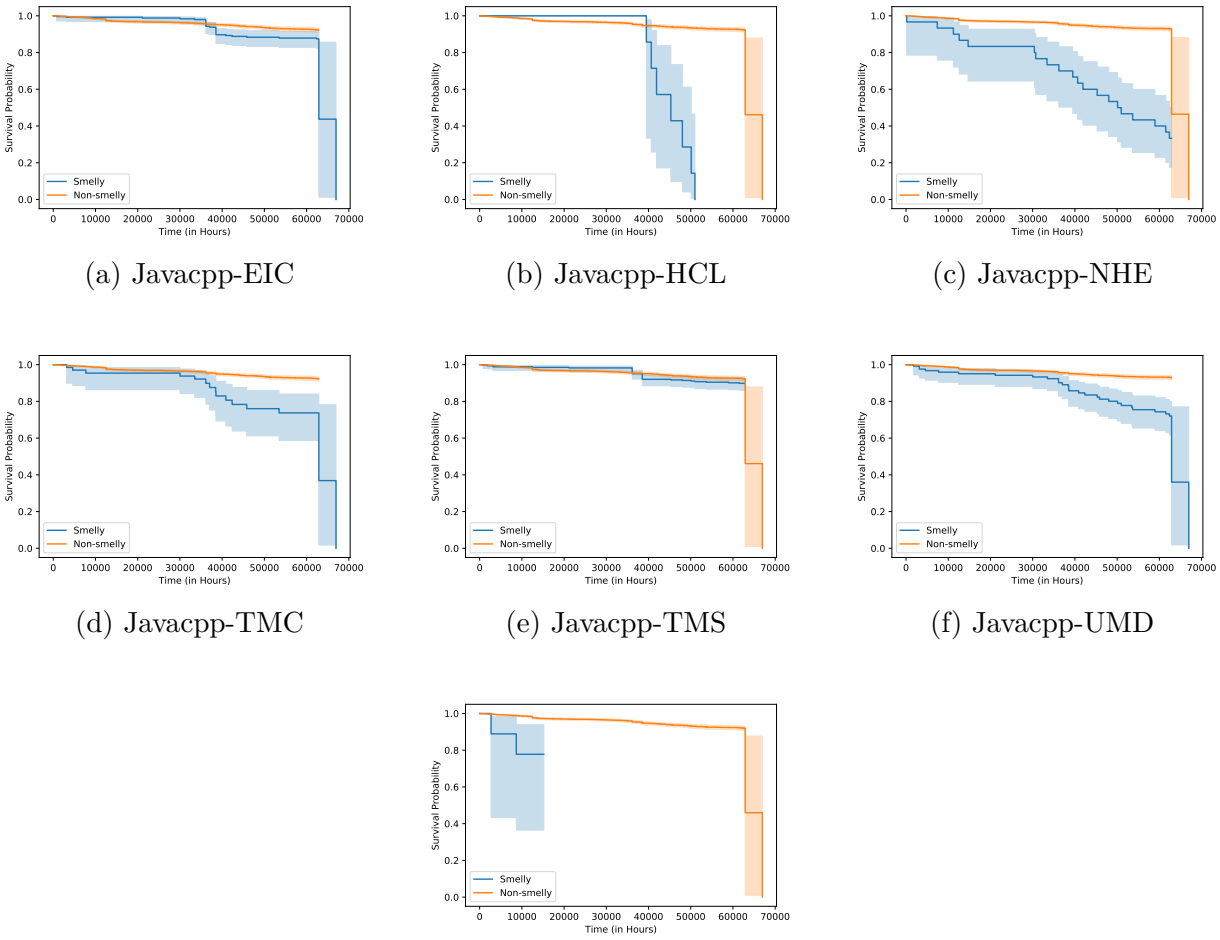
(a) Frostwire-ASRV (b) Frostwire-EIC (c) Frostwire-NHE

(d) Frostwire-TMC (e) Frostwire-NSL (f) Frostwire-TMC

(g) Frostwire-UMD (h) Frostwire-UMI (i) Frostwire-UP

Figure 10.4 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Frostwire

proportion of 10.17%, 8.11%, 6.95%, respectively. We assigned N/A for the design smells *Excessive Objects* and *Not Caching Objects* because we did not find occurrences of these smells in our dataset. The 0% means that we did not find occurrences of that specific smell in that bug category. 35.9% of the buggy files with occurrences of the smell *Unused Parameters* are related to the category of platform and dependencies, and 24.07% are related to communication and network issues. From our analysis, we did not find any occurrences of the design smell *Excessive Inter-language Communication* in multi-language smelly files containing bugs related to database issues. 23.68% of the files with occurrence of the design smell *Unused Method Declaration* that experienced a bug are related to database issues, closely followed by concurrency bugs with proportions of 22.52% and 22.49%, respectively.



(a) Javacpp-EIC (b) Javacpp-HCL (c) Javacpp-NHE

(d) Javacpp-TMC (e) Javacpp-TMS (f) Javacpp-UMD

(g) Javacpp-UP

Figure 10.5 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Javacpp

26.92% of the buggy files with occurrences of the design smell *Too Much Scattering* experience bugs related to concurrency, 10% of the buggy files with occurrences of the design smell *Too Much Scattering* experience bugs related to libraries and features support.

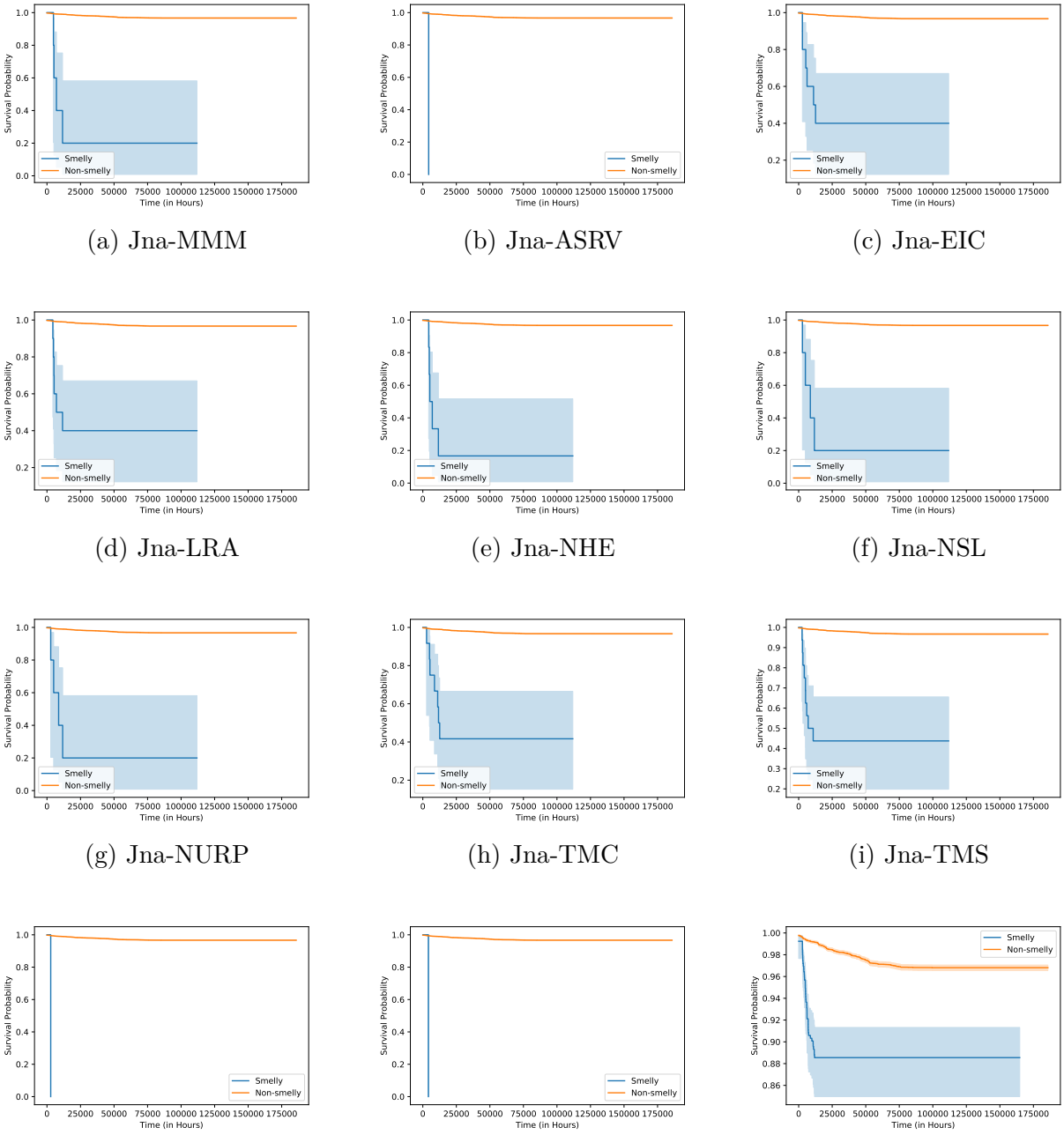
As in Table 10.8, 22.49% of smells related to *programming errors* are of type smell *Unused Method Declaration* closely followed by *Excessive Inter-language Communication* (19.40%), and *Unused Parameters* (16.45%). For the category *libraries and features support*, *Excessive Inter-language Communication* frequently occurs (28.03%). The design smells *Excessive Inter-language Communication*, *Memory Management Mismatch*, *Local References Abuse* are the most dominating smells related to *memory* issues with respective proportion of 23.61%, 21.59%, and 20.56%. In fact, *Memory Management Mismatch* and *Local References Abuse* are

related to the allocation and release of memory. The design smell *Excessive Inter-language Communication*, results in extra calls between the host and foreign language which will induce the allocation of memory to each object that is passed from one language to the other. Thus, not releasing the memory and increasing the inter-language communications could lead to *memory* issues.

Listing 10.1 presents an example of bug-fixing commit in a smelly file extracted from *Rocksdb* (options.cc).[3] The `'+'` signs in this code example shows the lines that were added during the bug-fixing commit, while the `'-'` sign shows the line that was deleted. This bug-fixing commit was assigned to the bug category *memory* issues based on the bug-fix commit message `"Fixed various memory leaks and Java 8 JNI Compatibility"`. In this example code snippet, we see that the memory leak was the result of using `GetIntArrayElements` to access a Java array without releasing the memory after using `ReleaseIntArrayElements` (design smell *Memory Management Mismatch*). As we discussed in Chapter 7, JNI treats Java objects and classes as reference types because of the type incompatibility between Java and C/C++. To access such reference types, JNI offers predefined methods. Those methods allocate memory to each element that is accessed. Therefore, in such cases developers should release the memory after usage. If the memory usage of those types is not managed correctly, allocating memory for new reference objects may fail [11,49]. Occurrences of this design smell may also lead to memory leaks as presented in Listing 10.1. Thus, we believe that developers should be cautious about files with multi-language design smells, especially *Unused Method Declaration*, *Excessive Inter-language Communication*, *Unused Parameters*, *Not Handling Exceptions*, *Memory Management Mismatch* and *Local References Abuse* because they are more prone to different types of bugs and may incur additional maintenance efforts.

Summary of findings (RQ10.4): The design smells *Excessive Inter-language Communication*, *Unused Method Declaration*, *Unused Parameters*, *Not Handling Exceptions* are the most dominant types of smells associated with different bug categories, especially *programming errors* and *libraries and features support* bugs. *Excessive Inter-language Communication*, *Memory Management Mismatch*, and *Local References Abuse* are the most dominating types of design smells related to *memory* bugs.

[3] `https://github.com/facebook/rocksdb/commit/c6d464a9da7291e776b5a017f0a5d33d61f2518b`



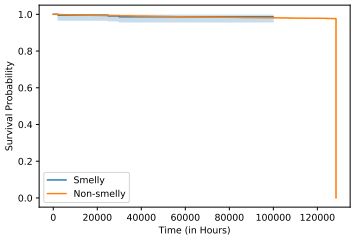
(a) Jna-MMM (b) Jna-ASRV (c) Jna-EIC

(d) Jna-LRA (e) Jna-NHE (f) Jna-NSL

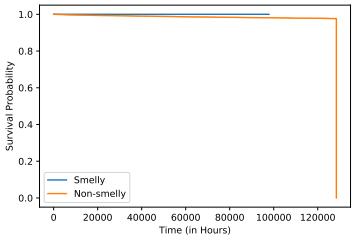
(g) Jna-NURP (h) Jna-TMC (i) Jna-TMS

(j) Jna-UMD (k) Jna-UMI (l) Jna-UP

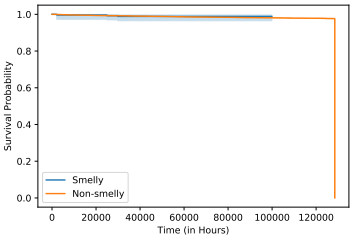
Figure 10.6 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Jna



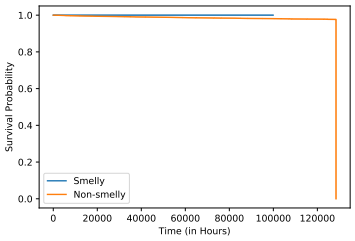
(a) OpenDDS-ASRV



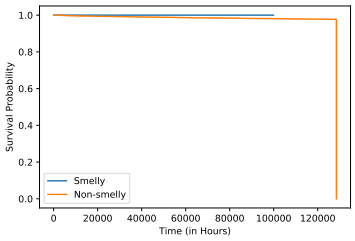
(b) OpenDDS-LRA



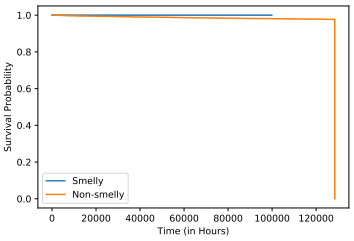
(c) OpenDDS-NHE



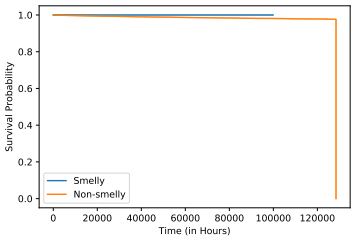
(d) OpenDDS-NSL



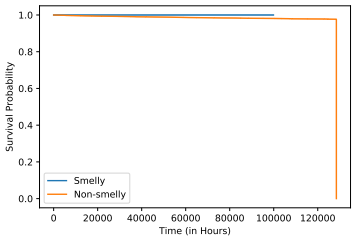
(e) OpenDDS-NURP



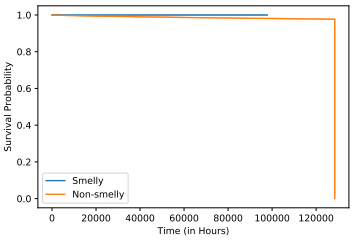
(f) OpenDDS-TMC



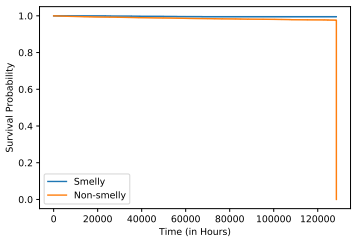
(g) OpenDDS-TMS



(h) OpenDDS-UMD

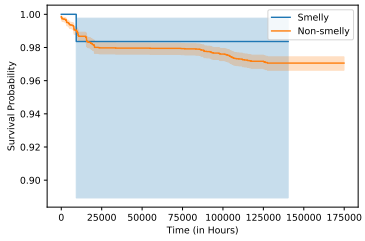


(i) OpenDDS-UMI

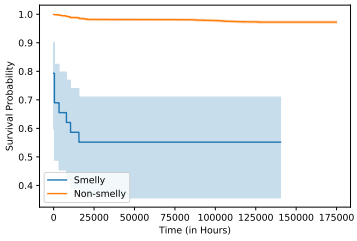


(j) OpenDDS-UP

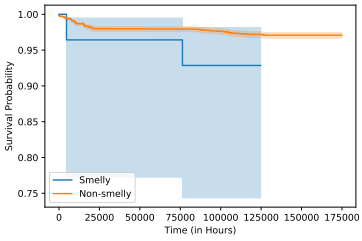
Figure 10.7 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - OpenDDS



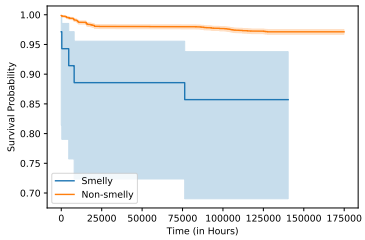
(a) PlJava-EIC



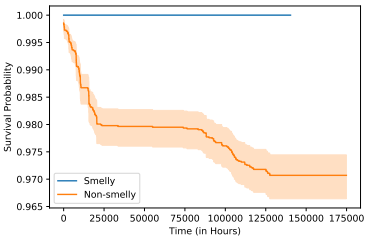
(b) PlJava-LRA



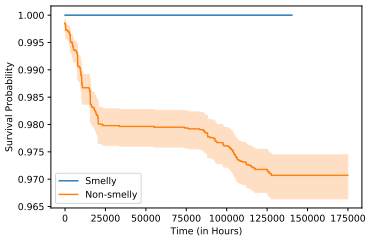
(c) PlJava-MMM



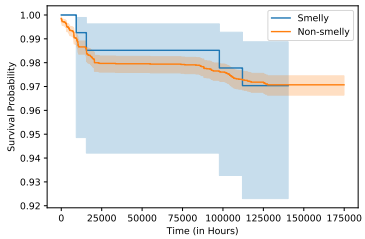
(d) PlJava-NHE



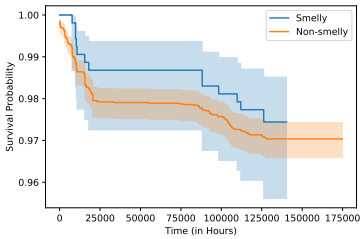
(e) PlJava-NSL



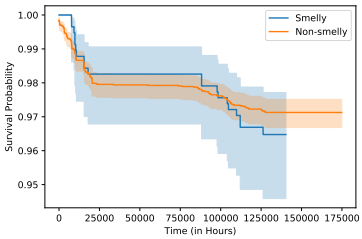
(f) PlJava-NURP



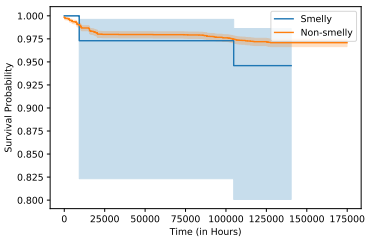
(g) PlJava-TMC



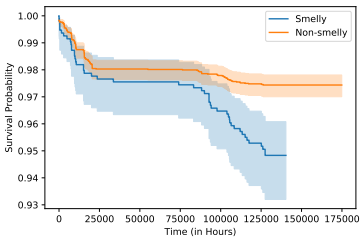
(h) PlJava-TMS



(i) PlJava-UMD

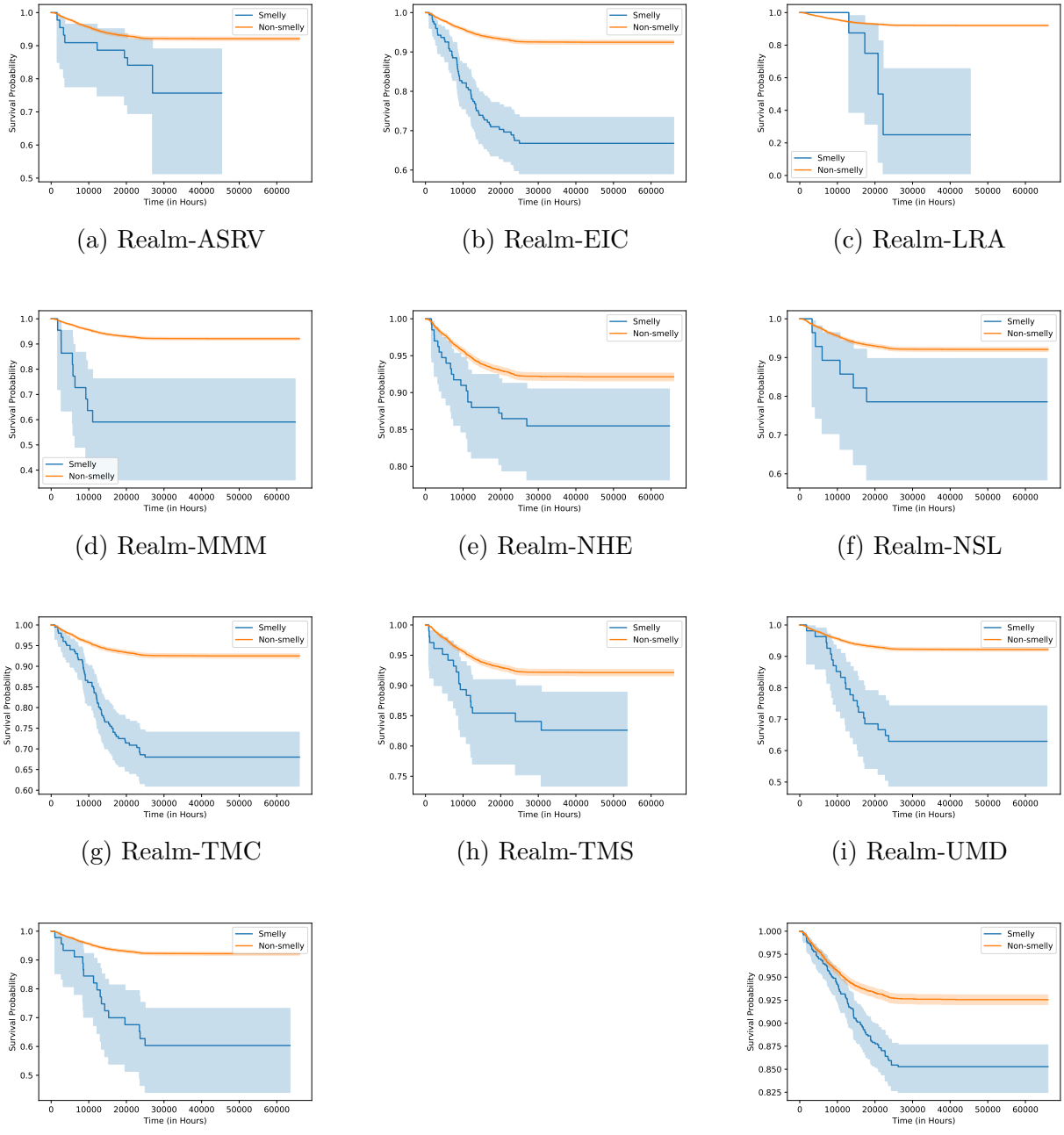


(j) PlJava-UMI



(k) PlJava-UP

Figure 10.8 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - PlJava



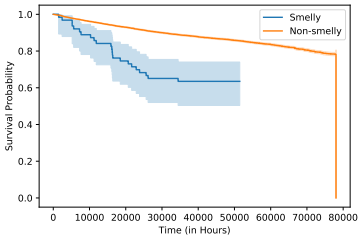
(a) Realm-ASRV (b) Realm-EIC (c) Realm-LRA

(d) Realm-MMM (e) Realm-NHE (f) Realm-NSL

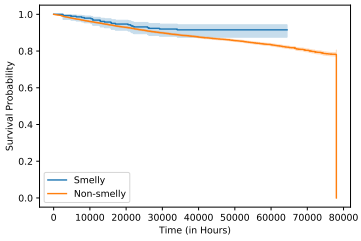
(g) Realm-TMC (h) Realm-TMS (i) Realm-UMD

(j) Realm-UMI (k) Realm-UP

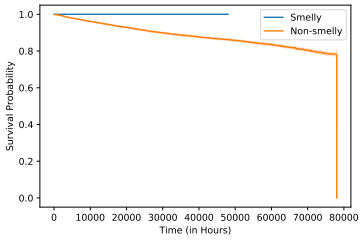
Figure 10.9 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Realm



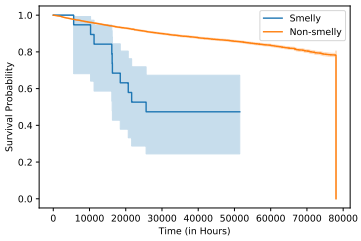
(a) Rocksdb-ASRV



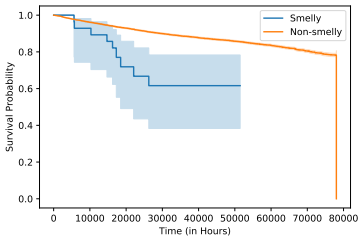
(b) Rocksdb-EIC



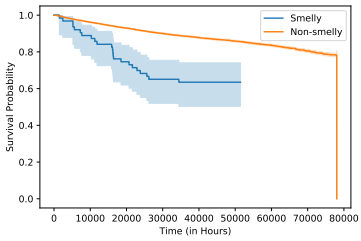
(c) Rocksdb-HCL



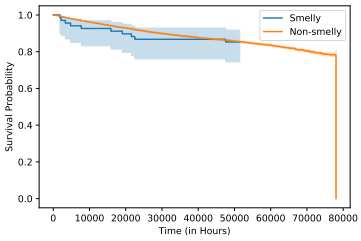
(d) Rocksdb-LRA



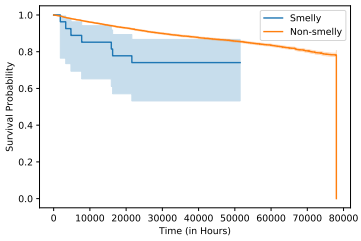
(e) Rocksdb-MMM



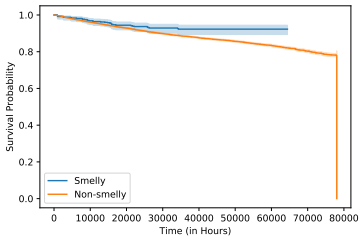
(f) Rocksdb-NHE



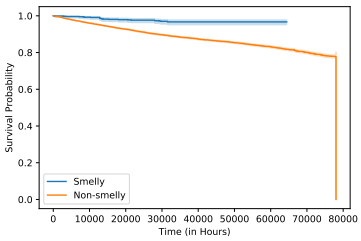
(g) Rocksdb-NSL



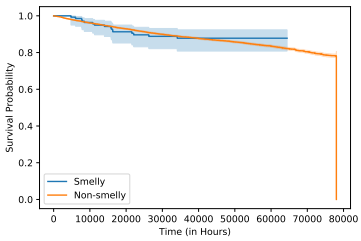
(h) Rocksdb-NURP



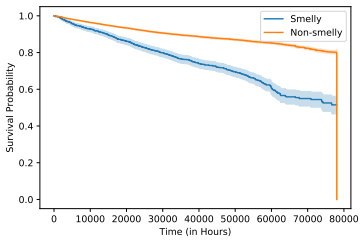
(i) Rocksdb-TMC



(j) Rocksdb-TMS



(k) Rocksdb-UMD



(l) Rocksdb-UP

Figure 10.10 Survival Curves for Bug-occurrence in Smelly and Non-smelly Files for Specific Types of Multi-language Smells - Rocksdb

Table 10.8 Distribution of Multi-language Design Smells Among the Bug Categories

Bug Categories↓/Smells→	UP	UMD	TMS	TMC	UMI	ASRV	NHI	EIC	NHE	NCO	NSL	HCL	NURP	MMM	LRA
Programming Errors	16.45%	22.49%	5.70%	8.97%	1.78%	1.92%	N/A	19.40%	7.37%	N/A	0.86%	1.90%	3.42%	6.62%	3.14%
Libraries and Features Support	17.19%	18.21%	10.93%	10.86%	1.91%	0.037%	N/A	28.02%	0.87%	N/A	1.97%	5.47%	1.046%	2.93%	0.56%
Memory	12.84%	10.55%	4.75%	2.59%	0.031%	1.56%	0%	23.61%	1.56%	N/A	0%	0%	0%	21.95%	20.56%
Platform and Dependencies	35.90%	13.92%	1.83%	2.56%	3.27%	0.37%	N/A	19.05%	0.73%	N/A	1.10%	7.69%	3.30%	4.03%	6.237%
Communication and Network	24.07%	13.75%	5.25%	7.15%	0.75%	7.70%	N/A	11.39%	15.69%	N/A	0%	0.37%	0%	6.94%	6.97%
Concurrency	21.52%	22.52%	26.92%	4.49%	0%	3.15%	N/A	5.63%	4.17%	N/A	0%	0%	0%	2.5%	9.10%
Database	13.16%	23.68%	0%	18.42%	0%	10.53%	N/A	0%	34.21%	N/A	0%	0%	0%	0%	0%
Performance	6.56%	7.63%	9.50%	0.53%	0%	0.13%	N/A	43.50%	16.73%	N/A	0.67%	0%	0.67%	8.97%	5.087%
Average	18.46%	16.59%	8.11%	6.95%	0.97%	3.17%	N/A	18.82%	10.17%	N/A	0.58%	1.92%	1.05%	6.74%	6.45%

Acronyms: UP: UnusedParameter, **UM:** UnusedMethodDeclaration, **TMS:** ToomuchScattering, **TMC:** Toomuchclustring
ASR: AssumingSafeReturnValue, **EO:** ExcessiveObjects **EILC:** excessiveInterlangCommunication, **NHE:** NotHandlingExceptions
NCO: NotCachingObjects, **NSL:** NotSecuringLibraries, **HCD:** HardCodingLibraries, **NURP:** NotUsingRelativePath
UMI: UnusedMethodImplementation,**LRA:** LocalReferencesAbuse,**MMM:** MemoryManagementMismatch

10.4 Discussion and Implications

This section discusses the results reported in Section 10.3.

10.4.1 Survival of Files with Multi-language Smells from Bugs

From our results presented for **RQ10.1**, we observe that multi-language smells have negative impact on the time to bug occurrences in smelly files. In seven out of the eight studied systems, bugs in multi-language smelly files occur faster than in files without those smells. In fact, our results show that the survival probability of files with occurrences of multi-language design smells is lower than the survival of files without multi-language design smells. Therefore, the studied design smells seem to negatively impact the software fault-proneness. This impact is similar to the impacts of mono-language design smells that have been widely studied in the literature and were reported to negatively impact systems by making classes more change-prone and bug-prone [23, 24]. Since multi-language systems are more complex and introduce additional challenges, such design smells are expected to increase the maintenance overhead and the risk of bugs related to these systems.

Results of **RQ10.2** show that some specific types of smells are more related to bugs than the others. The design smell *Not Handling Exceptions* is found to have a higher impact on fault-proneness (higher hazard ratio) compared to other types of multi-language design smells. The exception handling is particularly helpful to identify and report errors occurring in the code. Although the exception handling is quite simple in Java, it gets trickier when combining Java with the native code. The key reason is that the handling of exception differs depending on the programming language. In the native code, the exception handling remains pending until the control returns back to the Java code. Thus, developers should be cautious when handling the native exception, *e.g.,* example of bug-fixing commit from realm `"Fix helper class for throwing java exception from JNI"`). The same goes for *Assuming Safe Return Value* that should be used to make sure that the program was correctly executed. When an exception is thrown in Java, the control directly seeks for an immediate catch block to properly handle the exception. However, the scenario is quietly different with the native code. When an exception is thrown in the native code, answering such exception is postponed until the control returns back to Java code. Therefore, when using JNI, developers should write their own code for implementing the correct control flow for checking return values and handling and clearing exceptions to ensure the correctness of the program.

Moreover, our results show that the design smells *Local References Abuse* and *Memory Man-*

agement Mismatch also have a negative impact on the time to bug occurrence. These two smells point out issues related to the allocation and release of memory. Java objects are handled by the native code as reference types (*e.g.,* String, Class, Object), and predefined methods are used to access fields and methods. Such methods allocate memory that should be released after usage to avoid memory and security issues. Similarly, all native methods that return a Java object create local references in the reference table. The number of local reference is limited and exceeding that number will lead to memory leaks.

We believe that developers should be cautious when dealing with files containing the studied design smells because such files seem to have a lower survival probability before the occurrence of a bug. In fact, smelly files are more likely to be subject to bugs and may incur additional maintenance efforts.

10.4.2 Categories of Bugs occurring in Multi-language Smelly Files

As multi-language development involves combining programming languages with different semantics and lexical rules, this can often complicate code comprehension, and negatively impact maintenance, because of bugs occurrences. Although all the bug topics and categories presented in this dissertation are important. Our results in **RQ10.3** show that *programming errors*, *libraries and features support*, *memory* are the most dominating categories of bugs related to multi-language smelly files. Thus, we recommend that researchers and developers pay more attention to all the types of bugs discussed in this dissertation, especially the bugs related to *programming errors*, *libraries and features support*, and *memory*. Previous studies that investigated the types of bugs in multi-language systems also reported that *programming errors* and *memory* issues are among the most common types of bugs [3,17]. The heterogeneity of components in multi-language systems could lead to programming errors. In fact, each programming language has its own rules (*i.e.,* semantic, lexical, and syntactical), thus, generic programming errors could easily occur. This category of bugs includes program rules compatibility issues, such as the differences between the Java and native code regarding the management of exception, native return types, etc. It also includes the bug type breaking changes *i.e.,* code changes in one part of the software system that may potentially cause related components to fail either at compile-time or run-time[4]. Indeed, tracking code dependencies across components written in different programming languages could be a challenging task (*e.g.,* from rocksdb `"Breaking updates Conflicts: .JNI-h-file-generation.launch"`). Our results also highlight bugs related to *libraries and features support*. This category includes the integration of existing libraries for reuse, which is one of the main expected benefits of

[4]`https://codingforsmarties.wordpress.com/2017/04/02/breaking-changes/`

multi-language development. Several articles and developers' blogs discussed issues related to the integration of external libraries into JNI as well as their dependencies (*e.g.,* bug-fixing commit message from conscrypt `"add missing libraries to JNI lib Conscrypt: fixing Android.mk dependencies Initial empty"`)[5].
Therefore, developers should use external libraries with caution and manage all the related dependencies carefully, to take advantage of the reuse of code.

We reported in **RQ10.4** that almost all the types of the studied design smells are quite distributed among all the bug categories. From our results we found that the design smell *Excessive Inter-language Communication* is the most frequently occurring smell type in all the bug categories with an average value of 18.82%. Multi-language systems are complex by nature and having excessive communication between components written in different programming languages may increase the complexity related to such systems and consequently may results in different types of bugs. The design smells *Excessive Inter-language Communication* frequently occurs in files that experienced bugs related to *performance*. This could be explained by the extra overhead that could result from the excessive calls between the Java and the native code that define the nature of this design smell type. In fact, context switching from the Java environment to the native code may be time consuming[6]. The design smell *Excessive Inter-language Communication* also occurs frequently in files that experienced bugs related to *memory*. Such bugs could be related to the fact that passing Java objects to native code requires the call to some predefined methods to access and manipulate the data. Such methods allocate memory that should be released after usage. Indeed, the JNI creates references for all Java object arguments passed in to native methods, as well as all objects returned from JNI functions. Therefore, excessive calls between the Java and native code may lead to *memory* issues if not handled correctly. Design smells related to unused code *i.e., Unused Parameters* and *Unused Method Declaration* are also highly distributed among all the bug categories presented in this Chapter with average values of 18.46% and 16.59% respectively. This could be due to challenges introduced by the design smells *Unused Parameters* and *Unused Method Declaration*due to the unused code. These smell types even if not directly related to bugs, could increase the maintenance effort and impact the code comprehension and maintainability, which may lead to the occurrences of bugs. Our results also highlight that the design smell *Not Handling Exceptions* is the most frequently occurring smell type in all the bug categories except the bugs related to *database* issues. This type of design smells was previously reported as related to bugs [12, 82, 129]. A bug related to this smell type was also reported

[5] `https://www.databasedevelop.com/article/12233882/How+to+resolve+dll+dependency+with+external+library`

[6] `https://www.tutorialfor.com/blog-219186.htm`

in *Conscrypt*: `"Fixed various memory leaks and Java 8 JNI Compatibility WARNING in native method: JNI call made without checking exceptions when required to from CallObjectMethod"`). Developers were not checking for Java exceptions after calling JNI methods. As explained in Chapter 2, the management of exceptions is not automatically handled in all the programming languages. Indeed, unlike for the Java code, the native code does not support the automatic exception handling. Therefore, developers that do not have enough knowledge about the exception handling mechanism in the JNI context could introduce bugs and other maintenance challenges.

From our results we also observed that some of the smell types with a higher risk of the introduction of bugs are also associated with bug categories with a higher proportions of bugs. Our results in **RQ10.3** show that bugs related to *programming errors*, *libraries and features support*, and *memory* are the most dominating categories of bugs related to multi-language smelly files. In **RQ10.4**, we found that the design smells *Unused Method Declaration*, *Excessive Inter-language Communication*, and *Unused Parameters* are the most dominant types of smells that exist in the bug category *programming errors* and *libraries and features support*. While, the most dominant types of smells in the bug category *memory* are *Excessive Inter-language Communication*, *Memory Management Mismatch*, and *Local References Abuse*. From **RQ10.2**, we report that these design smells are among the smell types posing higher risk of bugs considering the time to bug occurrence. Therefore, we believe that developers should be concerned about these types of design smells. Indeed, files with these types of design smells seem to not only have a low survival probability before the introduction of a bug, but they also appear to be among the most frequent types of smells in the most dominant bug categories.

10.4.3 Comparative Insights Regarding Previous Findings

Our results from Chapter 9 show that some types of design smells are more related with bugs than others: *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding Libraries*, *Not Handling Exceptions*, *Memory Management Mismatch*, and *Not Securing Libraries*. However, in that study, we studied the correlation between individual smell types and the introduction of bugs. In this study, we considered another perspective of analysis (*i.e.,* survival analysis) and studied how long smelly files survive before the occurrence of the event of interest (*i.e.,* bug occurrence in our case). Therefore, our results emphasize on the timeline and risk level of the bugs.

From the correlation analysis of multi-language design smell types and introduction of bugs we found that *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding*

Libraries, *Not Handling Exceptions*, and *Memory Management Mismatch*, and *Not Securing Libraries* design smells are observed to be more related to faults compared to other smells. From the survival analysis of smelly files until the introduction of a bug we found that files with design smell types *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, and *Unused Method Declaration* lead to bugs faster compared to files without those types of smells. The design smell types *Not Securing Libraries*, *Unused Method Implementation*, *Excessive Inter-language Communication*, *Too Much Clustering* and *Not Using Relative Path* were also in some cases appearing in the top 5 smells with higher risk of bugs. **From these results we can conclude that the following types of design smells are not only highly correlated with the introduction of bugs, but also that files containing them are more likely to experience bugs faster than files without them: *i.e., Unused Parameters*, *Not Handling Exceptions*, *Memory Management Mismatch*, *Too Much Clustering*, and *Not Securing Libraries***. The *Unused Parameters* and *Unused Method Declaration* design smells are related to unused parameters and classes with excessive number of unused native methods declaration respectively. Thus, the impacts of the unnecessary code resulting from these two smells could affect code maintainability and the comprehension of JNI systems, which may lead to the introduction of more bugs within a short period of time. Similarly, several articles and developers' blogs discussed bugs related to mishandling JNI exceptions and the management of the memory [11, 49, 82]. Therefore, it is not surprising to observe that the design smells *Not Handling Exceptions* and *Memory Management Mismatch* lead to bugs earlier compared to other types of design smells.

The design smell *Not Securing Libraries* was reported to be correlated with the introduction of bugs. This smell type also appears in the top five smells with higher risk of bugs. While files with occurrences of design smells *Local References Abuse*, *Unused Method Implementation* and *Assuming Safe Return Value* were not perceived to be highly correlated with bug occurrences as we reported in Chapter 9. In this study, we observe that files with these smell types experience bugs faster compared to some other types of smells that were found to be more correlated to bugs in Chapter 9.

10.5 Threats To Validity

In this section, we discuss potential threats to the validity of our study following guidelines for empirical studies [118].

Threats to construct validity concern the relation between the theory and the observation. We relied on the smell detection approach proposed in Chapter 8. The approach was reported to have a minimum precision and recall of 88% and 74%, respectively. We relied on the SZZ algorithm to identify bug-inducing commits. The heuristics used in SZZ may not be 100% accurate. However, it has been successfully used in multiple studies investigating design smells [24,107,151]. The heuristics for finding bug-fix commits using keywords may also introduce false positives. We mitigated this threat by using keywords that were reported to be associated with bug-fixing commits [132,158]. When analyzing the smelliness of files that experienced bugs, we considered the whole file as participating in the design smell. Hence, the design smell present in the file could be in different code lines than the bug. A possible threat is related to the manual labeling of bug topics. We are aware that in some cases, developers might not have provided in the commit messages all the details related to bug or might have used some abbreviations. Therefore, the retrieved bug topics may not be 100% accurate. We reduced the threat related to the manual labeling of bug topics by relying on both keywords and selected commit messages and combining both manual and automatic approaches. Moreover, such methods have been applied in previous studies [149, 150]. The list of bug topics may not be exhaustive and might not reflect all the bugs related to multi-language smelly files. However, we are reporting our observations on possible bug topics that could be related to smelly files. Further investigation with a larger data set could lead to an exhaustive list of multi-language bug topics.

Threats to internal validity. We do not claim causation rather we report observations and explain our findings. This study is an internal validation of multi-language design smells that we previously defined and presented in Chapter 7 [34,49]. Therefore, the subset of multi-language design smells that we are considering may present a threat to validity. However, this catalogue was published in a pattern conference and went through rounds of shepherding process and Pattern Writers' Workshop in which we received meaningful comments to refine and improve the patterns before being validated and published.

Threats to conclusion validity concern the relationship between the treatment and the outcome. In this study, we were careful to acknowledge the assumptions of each statistical test that we used.

Threats to external validity address the possibility to generalize the results. We limited the scope of this study to open-source projects. However, the subject systems represent different domains and project size. We studied a particular subset of multi-language design smells

(JNI). Thus, further validation with other sets of languages would give more opportunities to generalize the results.

Threats to reliability validity To mitigate this threat we provide in this dissertation all the details needed to fully replicate our study. We used open-source projects available in GitHub[7].

10.6 Chapter Summary

In this Chapter, we examine the impact of multi-language design smells on the time to bug occurrence. We collected occurrences of 15 multi-language design smells on eight open source projects using the approach presented in Chapter 8. We analyzed a total of 270 snapshots. We performed a survival analysis and compared the time until a bug occurrence in multi-language files with and without the studied smells. We performed a topic modeling followed by a manual investigation to capture the categories and characteristics of bugs in files with multi-language smells. Our results show that multi-language smelly files experience bugs faster than files without those smells. Also, multi-language smells are not equally bug-prone. Developers should consider giving a special attention to files containing *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, and *Unused Method Declaration* design smells. Programming errors, libraries and features support, and memory issues are the most dominant types of bugs in multi-language smelly files. Our investigation reports several findings that we hope will raise practitioners' awareness about the impacts of different types of multi-language smells, while helping them prioritize their maintenance tasks.

[7] `https://github.com/ResearchMLS/Survival_Analysis`

CHAPTER 11 DEVELOPERS' PERCEPTION OF MULTI-LANGUAGE DESIGN SMELLS

11.1 Chapter Overview

From our results in Chapter 9, we found that the multi-language design smells presented in Chapter 7 are prevalent and that they have a negative impact on the software fault-proneness. In addition, in Chapter 10, we found that multi-language smelly files experience bugs faster than files without those smells. Our results from the last two chapters also highlight that the impact vary from one specific type of smell to the other. However, these assertions have never been verified with professional developers. Therefore, we aim through this chapter to complement the results presented in Chapter 9 and Chapter 10 and assess the relevance of the studied design smells from the developers' perspective. We also aim to investigate the perceived severity and impacts of multi-language design smells on software quality attributes. To achieve our objectives we extracted the information of the developers that contributed to the files impacted by the design smells studied in Chapter 10. We then designed two surveys. An open survey targeting professional software developers in general, but also a closed survey targeting developers that contributed in the smelly files. We decided to conduct a survey with professional developers because they are the one who interact daily with multi-language systems and suffer from their challenges.

11.2 Study Design

We present in this section our methodology to perform this study.

11.2.1 Setting Objectives and Research Questions

We started by setting the objective of our study. Our objectives are to assess the perceived relevance of multi-language design smells and their severity. We chose to carry out an empirical study using a survey because development and maintenance are manual activities

Part of the content of this chapter is published in:
Mouna Abidi, Moses Openja, Foutse Khomh, "Multi-language Design Smells: A Backstage Perspective", *Proceedings of the 17th International Conference on Mining Software Repositories (MSR), 2020*, and
Mouna Abidi, Md Saidur Rahman, Moses Openja, Foutse Khomh, "Multi-language Design Smells: A Backstage Perspective", *accepted in principle for publication in Empirical Software Engineering (EMSE), 2021*

performed by the developers. Developers' perception of the relevance and impact of multi-language design smells is important since they are the ones who maintain the multi-language systems daily and who suffer from the associated challenges. In this study, the quality focus is source code comprehension and maintainability, which can be negatively impacted by the occurrences of design smells. The context of the study consists of (i) objects, *i.e.,* design smells detected in open source projects; and (ii) subjects (developers), *i.e.,* professional developers sharing their perception of multi-language design smells. Our target subjects in this study are software developers in general but also the original developers who contributed to the analyzed systems. We defined our research questions as follows:

RQ11.1: **To what extent do multi-language design smells reflect developers' perception of design problems?** We aim to understand how developers assess if the design smells described in Chapter 7 are considered by developers as a design or implementation problems. Our goal is to capture the opinions of software developers in general, but also the opinion of the specific group of developers who contributed to the code impacted by those smells. We defined the following null hypothesis H_1*: Developers do not consider multi-language design smells as a design or implementation problem.*

RQ11.2: **What are the perceived impacts of multi-language design smells on software quality?** We aim to study the perceived impact of the multi-language design smells described in Chapter 7, on a selected set of software quality attributes. These design smells were defined and cataloged based on several sources of information (*e.g.,* developers blogs, literature, bug reports, and source code) [34, 49]. In Chapter 9 and Chapter 10 we have observed that these design smells increase the risk of faults in software systems. In this research question, we aim to understand the perception of developers regarding the severity of the proposed smells. This is important, since the urgency with which they refactor the smells is likely to be affected by their perception of the importance of the issues posed by the design smells. We defined the following null hypothesis H_3*: Developers do not perceive the studied multi-language design smells as having a negative impact on software quality attributes.*

RQ11.3: **What are the design smells that developers perceive as the most harmful?** During maintenance activities, developers are interested to identify parts of the code that should be tested in priority or refactored. Hence, we aim to identify design smells that are perceived by developers as the most critical, *i.e.,* making the project more prone to faults or increase maintenance costs. We defined the following hypothesis H_4*: Developers perceive the design smells to have equal impacts on multi-language systems.*

RQ11.4: **Do developers plan to refactor multi-language design smells?** Design smells are generally associated with a specific set of refactoring strategies depending on the type of smell. Depending on the development and maintenance costs, developers may decide whether or not to remove some specific smells. Therefore, we aim to investigate to what extent developers would apply such refactoring. We defined the following hypothesis H_5*: There is no specific strategy applied by the developers to refactor and remove multi-language design smells.*

11.2.2 Study Context

To achieve our objectives and answer our research questions, we started by analyzing the source code of selected open source projects.

Material and Objects The objects considered in this study are the occurrences of 15 types of multi-language design smells detected in eight open source projects. We analyzed a total of 270 snapshots as presented in Chapter 10. To ensure reliability of our surveys and to mitigate any possible threats related to the recall and-or precision of the detection approach, we manually validated all the occurrences of design smells that were used in our survey questions to developers. This manual validation allowed us to remove any false positives. Note that, we did not retrieved all instances of the studied smells in each object system.

Participants The survey described in this study is a combination of both an open and a closed survey. We present in the following our methodology to gather responses from the participants *i.e.,* developers for each of those surveys. For the open survey, we collected developers' backgrounds and demographic information to deal with the representativeness of the results. We followed guidelines of sampling methods to build samples that seek to meet the goals of this study to ensure better representativeness [159, 160].

- **Open Survey:** We used a convenience sampling and randomly contacted developers that satisfy the criteria of the study. We used LinkedIn[1] as a research tool to reach potential developers. Similar to previous work, we combine different sampling strategies [160]. By convenience sampling we consider available developers willing to participate in the survey. Since in this study we present code snippets only for JNI systems, we target developers having experience with both Java and C/C++ programming languages.

[1] `https://www.linkedin.com/`

- **Closed Survey:** We reused the detection results of the study presented in Chapter 10. We also collected information about the developers that contributed to the smelly files based on the commit logs. By 'contributor' we refer to any developer who did at least a commit on those files. Note that in the closed survey, a developer is only asked about smells contained in the files that he contributed to.

We decided to run both an open and a closed survey to capture not only the perception of developers who contributed to the code of smelly files, but also the perception of other developers who are less familiar with the smelly code files.

Studies in the literature discussed the ethics of sampling strategies [160]. This study was subject to ethical approval from the Research Ethics Board of Polytechnique Montreal which regulates the ethical and scientific criteria designed to protect human participants[2].

11.2.3 Study Procedure

The experimental protocol consists of surveys that developers had to answer through the *CheckMarket* website[3]. We combine in this study both open and closed surveys[4]. The open survey is targeting software developers in general while the closed survey is targeting developers extracted from commit logs and identified as contributors in the smelly files. We prepared the surveys based on our literature review of the state-of-the-art on multi-language systems and design smells. This review helped us to identify the dimensions and scope of the questionnaire, the individual questions, and the possible answers for each question [61, 63, 71, 110].

Both surveys start with a preamble that includes the information about the principal investigator, the research team, the ethical rights and responsibilities of the investigators and developers (*e.g.,* policy of the study with respect to anonymity). The preamble also describes the objective of the study so that developers understand our motivations for this survey and have full information to decide whether or not to answer the surveys (or parts thereof). In the surveys, we also provide definitions of the concepts and the design smells used in this study.

The open survey consists of three parts, while the closed survey consists of two parts. As shown in Table 11.1, the open survey contains in the first part four closed questions to retrieve background information about the developers and their profiles. The survey asks about their

[2] `https://share.polymtl.ca/alfresco/service/api/node/content/workspace/SpacesStore/b7fbaa9e-8055-41cc-b016-dac345f6cb97?a=false&guest=true`

[3] `https://www.checkmarket.com/`

[4] `https://s-ca.chkmkt.com/?e=189517&h=B445656A9769E90&l=en`

work position, their number of years of working experience, the domain of activities of their organisations, and their levels of skills in selected programming languages. These languages were reported to be in the top ten list of languages used world-wide[5]. Note that we did not include the background information in the closed survey since we are not randomly contacting developers but our target population for the closed survey is the original developers who contributed in the object systems. The second part of the open survey (which corresponds to the first part of the closed survey) contains general questions about multi-language design smells. It asks about the perceived impact on a set of selected software quality attributes, and the perceived severity of each smell type. For the quality attributes, we selected the following four attributes discussed by Gamma et *et al.* in their seminal book about design patterns, *i.e.,* expandability, modularity, reusability, and understandability, and three other attributes, *i.e.,* simplicity, learnability, and performance. We selected these quality attributes because of their relevance for design (anti) patterns and smells [110, 161].

As shown in Table 11.1, we kindly asked the developers in the third part of the open survey (second part of the closed survey) to perform a subset of tasks (Section II of Table 11.1). We provided in those tasks, the source code snippets and asked the developers about whether or not the presented code snippets contain implementation or design problems, *i.e.,* the design smells. We also asked the developers whether they would consider or not to refactor such code. We proposed a refactored solution and asked the developers whether they would consider or not to apply that refactoring. We also injected code snippets that do not contain any of the design smells to limit the bias of this study, *i.e.,* to isolate situation in which developers may provide arbitrary answers or always indicate the existence of design smells. Following previous work [110], for each type of smell, we randomly selected a representative set of instances to perform the survey. Depending on the type of smells, an instance could be a method, a class, files, or a combination of source code files. We ask about a single smell at the time and do not present code affected by more than one design smell, since we want to evaluate each smell separately. Table 11.1 reports the survey questions.

The tasks generated in Section II for each type of smell in our survey (as presented in Table 11.1) resulted in an important number of survey questions (15 design smells tasks and 3 cleaned code tasks, *i.e.,* not impacted by design smells). Therefore, we decided to divide the questions of the open survey as follows: we kept the first part of the survey (Section I of Table 11.1), that asks about design smell types in general (without example of code), common to all the surveys. We decided to keep this part common to all the surveys because it contains general questions about multi-language design smell types and is used to answer specific

[5] `https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`

Table 11.1 Survey Questions

N.	Survey Question
Background (Only for the open survey)	
Q1	What is your role within your organization?
Q2	How many years of experience do you have in software engineering?
Q3	What is the domain of activity of your organization?
Q4	What is your level of skill in the following languages?
Section I - Multi-language Design Smells	
Q1	How do you evaluate the impact of those design smells on the following software quality attributes?
Q2	Please rank the following design smells from the harmful to the less harmful
Section II - Specific Tasks for Multi-language Design Smell	
Q1	In your opinion, does the following code fragment(s) contain any occurrence of design smell (implementation and-or design problem)?
Q2	If you answered yes to the previous question, please provide an explanation or specify the design smell(s) involved?
Q3	Would you apply the following refactoring or would you prefer to keep the initial implementation? please explain.

research questions *i.e.,* **RQ11.2 and RQ11.3**. We divided the tasks (Section II of Table 11.1) among two open surveys by selecting a subset of tasks (smells) for each survey. The second part of each of the open surveys includes from 7 to 8 tasks containing code snippets affected by occurrence of design smells. Each task refers to one type of design smells. It also includes 3 cleaned tasks. Table 11.2 provides an overview of the smell types and tasks associated with each survey type. The tasks are independent and were designed in a way that allowed us to use survey results even if a participant do not complete the survey until the end; *i.e.,* we can reuse the completed parts of the survey responses. Also note that for the tasks asking about the identification of the smell, all the questions are conditional, *i.e.,* if a participant reports that the proposed code do not contain any problem or design issue, he will be directly moved to the next task, if not he will be asked to answer the questions related to that task. For the closed survey, as mentioned earlier, a developer was only asked about smells contained in the files in which he contributed code. For all these reasons, the total number of answers considered in this study for the first part of the survey (Section I of Table 11.1) is higher than the total number of answers related to each task (Section II of of Table 11.1) as shown in Table 11.2.

We contacted a total of 500 developers through LinkedIn, we received 59 responses for the first open survey and 73 responses for the second open survey. For the closed survey we contacted a total of 237 developers, and received a total of 39 answers. Therefore, for the first part of the survey (Section I), we count a total of 171 answers, while for the other sections, the number varies from one specific type of design smell to the other because, we divided the survey tasks as described above. Also not all the developers responded to all the survey tasks as shown in Table 11.2.

This study protocol was submitted to Mining Software Repositories (MSR) conference. The protocol went through a review process and was evaluated by the committee prior to the

execution of the survey[6].

11.2.4 Data analysis

We present in the following our analysis method to answer the research questions:

For **RQ11.1**, following previous work [110], we computed for each design smell type, the percentage of answers in which the developers were able to:

- (1) Identify a design or implementation problem, *i.e.,* design smells when we asked them to evaluate the code snippet containing a design smell. By identification we consider the situation in which the developers selected 'yes' as response to the question: *In your opinion, does the following code contains any occurrence of design smell (implementation and-or design problem)?*;
- (2) Identify the specific smell that was introduced in the code snippet. By identification we consider a situation in which the developers provide a correct answer to the question: *If you answered yes to the previous question, please provide an explanation or specify the design smell(s) involved?*

The percentage of correct identifications was computed out of the total number of answers to capture the perceived relevance of each type of smell from the catalogue presented in Chapter 7. Our goal is to investigate whether the developers perceive the studied design smells as design or implementation problems, and if the perception differs from one type to the other.

For **RQ11.2**, we analyzed answers to the closed question: *How do you evaluate the impact of those design smells on the following software quality attributes?*. We computed for each type of smell and each quality attribute the percentage of answers in which the developers reported an impact of the smell on that quality attribute. Since the question used the following scale: Positive, Neutral, Negative, and Not applicable (N/A). We have for each smell, (i) the list of quality attributes that are perceived to be negatively impacted by each smell, (ii) the list of quality attributes that are perceived to be not impacted (neutral impact), and (iii) the list of quality attributes, if any, that could be positively impacted by those design smells according to the developers.

For **RQ11.3**, we analyzed developers' answers to the question: *Please rank the following design smells from the harmful to the less harmful.* Similar to previous work [29], we rely on Borda count to answer this research question. Borda count is a rank-order aggregation technique. If there are n candidates to rank (*i.e.,* design smells in our situation), the first

[6] `https://osf.io/6yqv5/?view_only=4cca6dc961b44303833917c236e2d667`

Table 11.2 Overall Developers' Results For the Smells Identification (Open and Closed Surveys)

Smell	Op1	Op2	Con	Fro	JNA	Jav	Ope	Plj	Rea	Roc
NHE	57	0	5	2	0	0	4	3	5	7
UP	0	63	5	2	1	0	4	3	5	7
NSL	58	0	5	2	1	4	4	0	5	7
ASRV	0	71	5	2	0	0	4	0	5	7
LRA	59	0	4	0	1	0	4	0	0	7
NURP	0	71	0	2	1	0	3	0	0	3
MMM	57	0	4	0	1	0	0	3	5	5
HCL	0	72	0	0	0	0	0	0	0	0
EO	57	0	0	0	0	0	0	0	0	0
NCO	0	69	0	0	0	0	0	0	0	0
TMC	55	0	4	2	0	4	3	3	3	4
TMS	0	71	0	0	0	4	4	3	3	4
EIC	0	67	3	2	1	3	0	2	0	3
UMD	0	70	3	2	0	3	4	2	3	4
UMI	54	0	0	1	0	0	3	2	3	0

Op1: OpenSurvey 1, **Op2**: OpenSurvey 2, **Con**: Conscrypt-ClosedSurvey
Fro: Frostwire-ClosedSurvey, **Jav**: JavaCpp-ClosedSurvey, **Ope**: OpenDDS-ClosedSurvey
Plj: PlJava-ClosedSurvey, **Rea**: Realm-ClosedSurvey, **Roc**: Rocksdb-ClosedSurvey

ranked candidates receive n points, the second-ranked receive n-1 points, etc. We obtain a list of all the design smells ranked from the most harmful ones to the less harmful.

For **RQ11.4**, we compute for each type of design smell presented to the developers (along side a refactoring solution), the percentage of answers in which the developers answered 'yes' to the question *Would you consider using this refactored solution or would you prefer to keep the initial implementation? please explain.*. We proposed three options to the developers *(i) yes, refactor with the proposed solution, (ii) refactor with an alternative solution, and (iii) no refactoring.* We also aggregate the answers to provide a general overview of developers' opinion on the application of the proposed refactored solutions.

11.3 Study Results

We now present the results of our survey answering our research questions.

Demographic information

As mentioned in the Section 11.2, we collected the demographic information only for the open surveys. From the open surveys we received a total of 132 developers' responses. Developers of the open surveys originated from different background. From our results we found that 40 (30.3%) are software engineers, 28 (21.2%) are developers, 16 (12.1%) are team leaders, 14 (10.6%) are project managers, 11 (8.33%) are architects, 8 (2.26%) are testers, five (3.79%)

are QA-managers, four (2.26%) are self-employed, and 6 (13.53%) selected the option "Other". From our results we noticed that the developers that answered the survey have mainly from five to ten years of working experience (57 out of 132 developers (43.2%)). 43 (32.6%) developers have more than ten years of experience. 30 (22.7%) have from one to five years of working experience, while only two (1.52%) have less than one year of experience. This diversity in the background gives us confidence that we reached a diverse group of developers. From the survey responses, we observe that developers are working in different domains of activities. 38 (28.8%) developers are in research and development, 25 (18.9%) are in analytics (business, IT services, big data), 18 (13.6%) are in networking , 18 (13.6%) are in games, 17 (12.9%) are in robotics and embedded systems, 7 (5.3%) are in banking and insurance. Nine (6.82%) of the developers selected the option "Other". We asked the developers about their levels of skills if they are Novice, have Little Knowledge, Practical, Comfortable, or Expert in C, C++, C#, Java, and Python. The majority reported being expert or comfortable with Java, C, and C++, and comfortable with C#. For Python, most of the developers reported being comfortable or practical. Figure 11.1 summarizes developers' skills and experience with the selected programming languages.

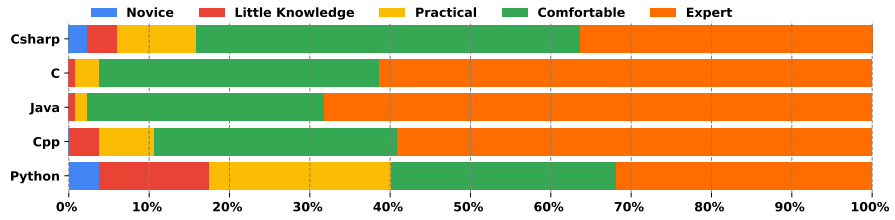


Figure 11.1 Developers' Experience With Programming Languages

RQ11.1: To what extent do multi-language design smells reflect developers' perception of design problems?

Approach Since our goal for this research question is to assess whether developers consider the design smells studied in this thesis as a design and–or implementation problems. For that, we proposed for each smell type an example of code snippet and asked the developers' whether they perceive any design or implementation problem. We also asked them to specify the design smell present in that code snippet.

Findings Figure 11.2a and Figure 11.2b provide the percentage of developers (for the open an closed surveys, respectively) who (1) reported that there was a design or implementation problem when we asked them to evaluate the code snippet containing a design smell, and

(2) correctly identified the smell that was contained in the code snippet, for all the types of smells studied in this thesis. We report in Table 11.3 the total number of responses where the smell was correctly identified by the developers for both open and closed surveys, and also the total number of smells that were not correctly identified. To give a comparative insight on the proportion of developers who correctly and incorrectly identified the smells, we calculate their corresponding percentages as shown in Table 11.3. We provide in the following the developers' responses grouped in two categories, *i.e.,* smells generally perceived as a design or implementation problem and smells that are not not generally perceived as a design or implementation problem.

Smells Generally Perceived and Identified as Design or Implementation Problems

Table 11.3 reports that for the design smell *Unused Method Implementation*, 87.95% (50) of the developers correctly identified the smell, while 12.05% (13) of the developers were not able to identify the design smell occurrence. For the design smell *Unused Method Declaration*, we report that 84.3% (69) developers were able to correctly identify the smell type, while 15.7% (22) were not able to identify the existing smell. For the design smell *Not Securing Libraries*, by analyzing Table 11.3, we see that 82.5% (70) of the developers correctly identified the smell, and 17.5% (16) were not able to identify the smell. For the design smell *Memory Management Mismatch*, our results show that 81.9% (63) of the developers were able to correctly identify the smell occurrence, and only 18.1% (12) were not able to identity the existing problem. For the design smell *Unused Parameters*, 77.8% (49) of the developers were able to correctly identify the smell occurrence, and 22.2% (14) were not able to identify the smell. For the design smell *Not Handling Exceptions*, Table 11.3 shows that 74.95% (64) of the developers correctly identified the smell *Not Handling Exceptions*, while only 25.05% (19) of the developers were not able to identify the design smell occurrence. For the design smell *Too Much Clustering*, Table 11.3 reports that 74.95% (56) of the developers correctly identified the smell, and 25.05% (22) were not able to identify the smell. For the design smell *Local References Abuse*, we report that 74.8% (56) of the developers were able to correctly identify the smell occurrence, while 25.2% (19) were not able to identify the smell. For the design smell *Assuming Safe Return Value*, our results show that 73.55% (69) were able to correctly identify the smell type, while 26.45% (25) were not able to identify the existing smell. For the design smell *Too Much Scattering*, our results show that 72% (64) of the developers correctly identified this design smell, while 28% (25) were not able to identify the smell occurrence. For the design smell *Excessive Inter-language Communication*, from Table 11.3, we see that 66.75% (37) of the developers correctly identified the smell, while 33.25% (37) were not able to identify the existing smell.

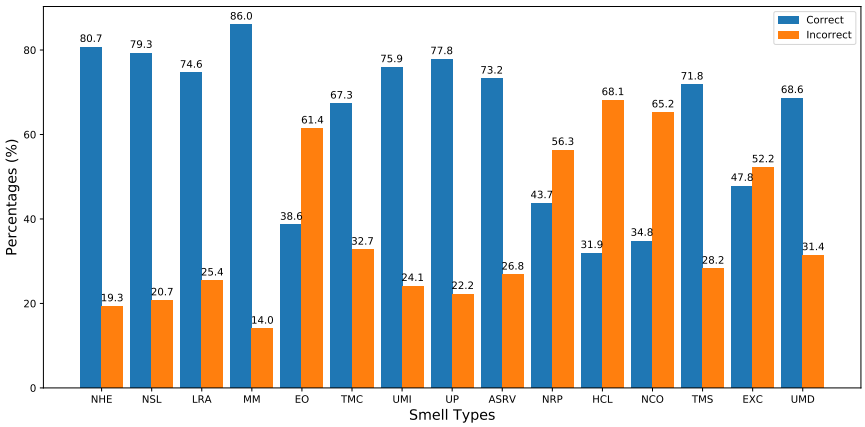
Table 11.3 Overall Developers' Results For the Smells Identification (Open and Closed Surveys)

Smell Name	#Correct	# Not Correct	%Correct	% Not Correct
NHE	64	19	74.95%	25.05%
NSL	70	16	82.5%	17.5%
LRA	56	19	74.8%	25.2%
MMM	63	12	81.9%	18.1%
EO	22	35	38.6%	61.4%
TMC	56	22	74.95%	25.05%
UMI	50	13	87.95%	12.05%
UP	69	21	75.95%	24.05%
ASRV	69	25	73.55%	26.45%
NURP	36	44	49.65%	50.35%
HCL	23	49	31.9%	68.1%
NCO	24	45	34.8%	65.2%
TMS	64	25	72%	28%
EIC	44	37	66.75%	33.25%
UMD	69	22	84.3%	15.7%

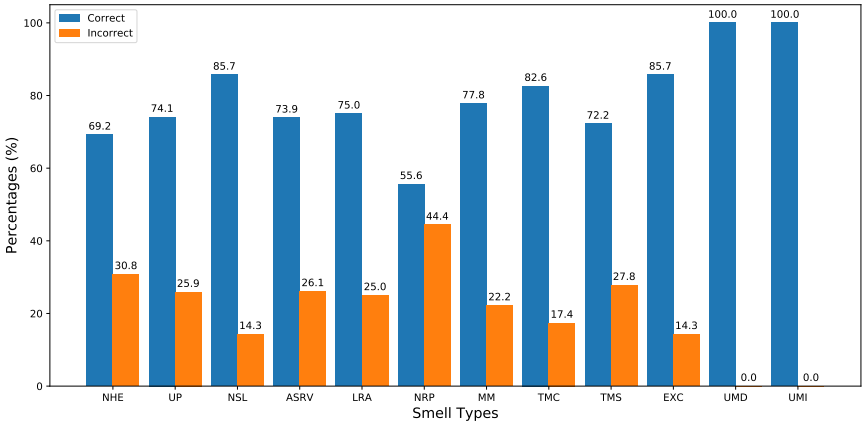
Smells Generally not Perceived as being Design or Implementation Problems

For the design smell *Hard Coding Libraries*, from analyzing Table 11.3, it appears that 31.9% (23) were able to correctly identify the smell type, while 68.1% (49) were not able to identify the smell. For the design smell *Excessive Objects*, we observe that 38.6% (22) of the developers were able to correctly identify the smell occurrence, and 61.4% (35) of the developers were not able to identify the design smell occurrence. For the design smell *Not Using Relative Path*, 49.65% (36) were able to correctly identify the smell type, and 50.35% (44) were not able to identify the smell. For the design smell *Not Caching Objects*, Table 11.3 reports that 34.8% (24) of the developers correctly identified the smell, while 65.2% (45) of the developers were not able to identify the design smell occurrence.

The results show that the design smells *Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, *Memory Management Mismatch*, *Unused Parameters*, *Not Handling Exceptions*, and *Too Much Clustering* were identified most frequently than the others. This could be explained by the nature of these types of design smells. The design smells *Unused Method Implementation* and *Unused Method Declaration* are respectively related to native methods that are implemented but never called, and native methods that are declared in the Java code but never implemented in the C/C++ code. These two types of design smells could be easily identified compared to other types of smells due to their simplicity. The design smells *Not Securing Libraries*, *Memory Management Mismatch*, and



(a) General Developers



(b) Original Developers

Figure 11.2 Developers' Perceived Relevance of Multi-language Design Smells

Not Handling Exceptions are also commonly discussed in developers' blogs [1]. The issues resulting from the *Memory Management Mismatch*, and *Not Handling Exceptions* are commonly discussed in the literature [11, 82]. All these reasons, could explain why these types of design smells were identified more frequently than the others. Also from analyzing commit messages we found a commit message related to the design smell *Unused Parameters* in *Conscrypt* (*"Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters"*). Therefore, the prevalence of these design smell types could explain their correct identification by the developers.

Summary of findings (RQ11.1): Overall, developers consider the proposed design smell to be reflective of design and implementation problems. Some specific types of smells were identified more frequently than the other types of smells. A majority of developers were able to correctly identify most of the design smells presented in this thesis, which provides an additional evidence of their relevance in multi-language systems.

RQ11.2: What are the perceived impacts of multi-language design smells on software quality?

Approach To assess the potential impacts of multi-language design smells, we asked the developers about their perception of the impacts of the studied multi-language design smells on selected software quality attributes. For each design smell type, we asked the developers to evaluate whether it presents a negative, neutral, or a positive impact on the specified quality attributes. The developers were given the option 'N/A' in case they were not able to evaluate the impact. We computed the percentages of responses, where each participant reported that a design smell of type i have a an impact (*i.e.,* positive, negative, neutral, or N/A) on the quality attribute q as described in Section 11.2.

Findings We report in Table 11.4 the summary of the perceived impact of multi-language design smells on software quality attributes, *i.e.,* expandability, modularity, reusability, understandability, simplicity, performance, and learnability. In general, the developers perceive the studied design smells as having a negative impact on the selected quality attributes. In the following, we discuss in details the results of each design smells for the top three quality attributes (based on the percentages of the developers' responses) that were perceived to be negatively impacted by the design smell.

Too Much Clustering From our results, 99.42% of the developers perceived a negative impact on the software simplicity. 95.32% and 93.57% of the developers reported respectively a negative impact on software understandability and learnability. This design smell adds complexity to the code and may impede the understandability and learnability as its implementation results in large classes with several native methods.

Unused Method Declaration From the analysis of the results in Table 11.4, it appears that 94.15%, 93.57%, and 88.89% of the developers reported a negative impact on the software expandability, simplicity, and understandability respectively. This design smell

introduce unused code as it is defined by native methods that are declared in Java code but never implemented in the native code. Such unused code may impede the software understandability. Indeed, it may be a challenging task for developers involved in a sub part of the system concerned by the smell, to clearly determine whether the smelly method is used by other components or not. Therefore, the expandability and reusability of components with occurrences of the design smell *Unused Method Declaration* may also be reduced.

Unused Method Implementation From our results, we can observe that 97.08% of the developers reported a negative impact on the software simplicity, 94.15% reported a negative impact on the understandability, while 92.4% of the developers reported that this design smell negatively impact the learnability. Similar to the design smell *Unused Method Declaration*, it is expected that unused code presents a negative impact on the software understandability. Given the purpose of this design smell, it is not surprising that it is considered to have negative impacts on the software simplicity and learnability.

Unused Parameters Table 11.4 shows that 91.81% of the developers reported a negative impact of this design smell on both software understandability and simplicity. 86.55% of the developers reported a negative impact on the software expandability. This design smell captures the fact that at least one of the native method parameters is not used within its native implementation. Therefore, such unused parameters may introduce some confusion and thus have negative impacts on the software understandability and simplicity.

Assuming Safe Return Value This design smell is reported by 87.72% and 85.96% of the developers, as having a negative impact respectively on the software modularity and learnability. 84.8% of the developers reported a negative impact on the software reusability. The perceived negative impacts could be related to the nature of this design smell. The modularity of the system is the degree to which the implementation of the functions are independent from one another. Therefore, not properly checking native return values may affect the software modularity and its reusability.

Memory Management Mismatch Table 11.4 reports that 87.13% and 83.63% of the developers reported a negative impact of this design smell on the software expandability and reusability, respectively. Our results also show that 77.19% of the developers perceive a negative impact of this design smell on the software modularity and its performance. This perception of the impact of this design smell could be explained by the purpose of the design smell. The JVM offers a set of predefined methods that could be used to access fields,

methods, and convert types from Java to the native code. Those methods allocate memory to each Java object that is used within the native code. Since Java's garbage collection system has no control over the use of dynamic memory in the native code. Therefore, developers' should release the memory allocated to each Java object. Not releasing the memory may introduce bugs and security issues [11,82]. Such issues may affect the component expandability and reusability.

Local References Abuse Table 11.4 shows that 93.57% of the developers reported a negative impact on the software reusability, while 86.55% and 85.96% of the developers reported a negative impact on the software expandability and modularity, respectively. Similarly to the design smell *Memory Management Mismatch*, this design smell type also may lead to memory issues [49]. JNI creates locale references for any Java object that is used in the native code. Not releasing local references may lead to memory issues, especially that the JVM enables to create a maximum of 16 local references. Exceeding the maximum number without informing the JVM may introduce bugs and memory leaks [11]. A component that is often subject to memory issues may have a low reusability and expandability. Especially that for JNI systems, several studies in the literature discussed the issues and vulnerabilities that could result from not properly releasing the memory [61, 82].

Not Handling Exceptions This design smell was perceived by 83.63% and 77.8% of the developers to have a negative impact on the software understandability and expandability, respectively. Our results show that 74.85% of the developers reported a negative impact on the software learnability. The management of exceptions has been discussed in the literature as one of the main concerns in the context of JNI systems [82]. Indeed, occurrences of this design smell may introduce some challenges and even bugs, which could explain the negative impacts reported by the developers. Since multi-language code requires to access from the host code components that are implemented in another programming language, mishandling exceptions may require additional effort to properly locate and fix the bug. Therefore, such design smell may negatively impact the understandability and learnability.

Excessive Inter-language Communication This design smell was reported by respectively 93.57% and 91.23% of the developers to negatively impact the understandability and modularity of the system. The results also show that 90.64% of the developers reported negative impacts of this design smell on the software learnability and simplicity. This design smell captures excessive communications between components written in different programming languages. Such excessive communications may impede the software understandabiliy,

learnability, and simplicity.

Excessive Objects Table 11.4 reports that 97.08% and 91.81% of the developers reported a negative impact on the software simplicity and expandability, while 90.65% of developers reported a negative impact on the software modularity. It is not surprising that this design smell is perceived as negatively impacting the software quality, since it occurs when developers pass objects from Java to native code, when only some of the object fields are needed. As explained in Chapter 8, the access to the Java object from the native code requires a call to specific methods following a specific order. Such methods may impact the simplicity and expandability of the software component.

Too Much Scattering Table 11.4 shows that 92.98% of the developers reported a negative impact on the software expandability and reusability, and 91.23% of the developers reported a negative impact on the understandability. The negative impacts could be explained by the fact that the smell occurs when the native declaration methods are scattered in different components of the Java code. This dispersion may impede the reusability of parts of the code and make it hard for developers to expand the code of the system.

From our results we also observe that some quality attributes are less impacted than the others. Table 11.4 shows that 70.18% of the developers reported that the design smell *Not Using Relative Path* has a neutral impact on the software simplicity. 69.59% and 67.84% of the developers reported a neutral impact on the software expandability and learnability. Giving the purpose of this design smell, having a neutral impact on the expandabiliy is rather surprising. However, the neutral impacts on the software simplicity and learnability could be explained by the definition of the design smell itself. Indeed, occurrences of the design smell *Not Using Relative Path* could result in a piece of code that is relatively simpler compared to it's refactored solution. However, this design smell may affect the reusability of the code and increase the cost of maintenance activities if the library is no longer available. The design smell *Not Securing Libraries* was also reported by some developers to have a neutral impact on some software quality attributes. 39.77% and 39.18% of the developers reported a neutral impact on the software understandability and expandability. Our results also show that 66.08% of the developers reported a negative impact of this design smell on the software reusability. The negative impacts of this design smells on software reusability could be expected. Indeed, insecure code leaves breaches in the code open to malicious attacks, and thus could reduce the resuability of that component. However, the neutral impact on the expandability is a bit confusing. The neutral impacts reported on the software understandability and expandability could be explained by the fact that this design smell

results on a simple block to load the library. Occurrences of this design smell are manifested when the native library is loaded without a call to any specific methods that ensure that the library could not be loaded by unauthorized code. To refactor this design smell, developers are required to add an additional security block, which may impede the effort required to understand the code.

The results in Table 11.4 show that in general the design smells are perceived to have a negative impact on the studied quality attributes. Understandability of the software was reported to be negatively impacted by the design smells. Understandability is one of the main concerns during maintenance activities. Several studies in the literature discussed the challenges of understanding and maintaining multi-language systems [1, 2, 5]. We believe that occurrences of design smells are likely to increase the effort needed to understand and maintain multi-language systems. From Table 11.4, we observe that the design smells *Too Much Clustering* (95.32%), *Unused Method Implementation* (94.15%), and *Excessive Inter-language Communication*(93.57%) are among the design smells mostly perceived to have a negative impact on the software understandability. Reusability is also reported to be negatively impacted by the design smells. The design smells *Too Much Scattering* (92.98%) and *Too Much Clustering* (90.06) are among design smells that are perceived to have the most negative impact on the software reusability. This may be the consequence of the negative impacts of the design smells on the software quality, *i.e.,* fault-proneness as reported in Chapter 9. *Too Much Clustering* smell occurs when too many native methods are declared in a single class creating a blob class resulting in poor comprehension and maintainability of the code. This is likely to add more risk of bugs as reflected in our study presented in Chapter 9. *Too Much Scattering* smell occurs when native methods scatters (often duplicated) over scarcely used classes which is likely to introduce maintenance challenges and increase the risk of bugs. Design smells may impact components' lifetime, limit their portability and impede their reusability. Developers considered a neutral impact on the studied quality attributes for some specific types of design smells *Not Using Relative Path*, *Not Securing Libraries*. These design smells were perceived to have a neutral impact on the simplicity. The simplicity was also reported to be negatively impacted mainly by *Too Much Clustering* (99.42%) and *Unused Method Implementation* (97.08%). This could be explained by the nature of these design smells, as they are adding extra complexity to the code and thus may affect its simplicity.

Summary of findings (RQ11.2): Most of the studied design smells were perceived as negatively impacting the studied software quality attributes. The design smell *Not Using Relative Path* was perceived to have a neutral impact. Most of the quality attributes were reported to be negatively impacted by the design smells.

Table 11.4 Perceived Impacts of Multi-language Design Smells

Smells	Impact	EXP	SIM	RUS	LRN	UND	PER	MOD
NHE	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	21.05	25.73	24.56	23.98	15.2	26,90	43.86
	Negative	77.78	73.1	74.27	74.85	83.63	71,93	54.97
ASRV	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	8.77	0.0	0.0	0.0	0.0	0.0
	Neutral	15.79	22.22	15.2	14.04	16.37	17,54	12.28
	Negative	84.21	69.01	84.8	85.96	83.63	82,46	87.72
EIC	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	14.62	9.36	12.87	9.36	6.43	13,45	8.77
	Negative	85.38	90.64	87.13	90.64	93.57	86,55	91.23
TMC	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	10.53	0.0	9.36	5.85	4.09	8,19	25.15
	Negative	88.89	99.42	90.06	93.57	95.32	91,23	74.27
TMS	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	7.02	14.04	7.02	12.28	8.77	14,62	13.45
	Negative	92.98	85.96	92.98	87.72	91.23	85,38	86.55
HCL	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	13.45	24.56	15.79	17.54	11.11	25,15	18.13
	Negative	86.55	75.44	84.21	82.46	88.89	74,85	81.87
LRA	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	11.7	0.0	0.0	3.51	0.0	0.0
	Neutral	12.28	12.87	5.26	21.64	19.88	14,04	12.87
	Negative	86.55	74.27	93.57	77.19	75.44	84,80	85.96
MMM	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	18.71	0.0	4.09	0.0	0.0	0.0
	Neutral	12.28	5.26	15.79	25.73	23.39	22.22	22.22
	Negative	87.13	75.44	83.63	69.59	76.02	77.19	77.19
NCO	N/A	2.34	2.34	2.34	2.34	2.34	2.34	2.34
	Positive	0.0	24.56	0.0	0.0	0.0	0.0	0.0
	Neutral	21.64	5.26	10.53	35.67	20.47	22.22	35.67
	Negative	76.02	67.84	87.13	61.99	77.19	75,44	61.99
NSL	N/A	1.75	1.75	1.75	1.75	1.75	1.75	1.75
	Positive	0.0	20.47	9.36	12.28	8.77	8.77	0.0
	Neutral	39.18	14.04	22.81	28.65	39.77	32,75	37.43
	Negative	59.06	63.74	66.08	57.31	49.71	56,73	60.82
NURP	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	7.02	0.0	2.34	5.26	3,50	0.0
	Neutral	69.59	70.18	40.35	67.84	59.65	59,06	16.37
	Negative	29.24	21.64	58.48	28.65	33.92	36,26	82.46
EO	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	8.19	2.92	11.7	13.45	9.94	13,45	9.36
	Negative	91.81	97.08	88.3	86.55	90.06	86,55	90.64
UMD	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	5.26	5.85	13.45	12.28	10.53	12,28	33.92
	Negative	94.15	93.57	85.96	87.13	88.89	87,13	65.5
UMI	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	14.04	2.92	17.54	7.6	5.85	9,36	8.19
	Negative	85.96	97.08	82.46	92.4	94.15	90,64	91.81
UP	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	13.45	8.19	14.04	36.84	8.19	19,3	16.96
	Negative	86.55	91.81	85.96	63.16	91.81	80,70	83.04

EXP: Expandability, **SIM:** Simplicity, **RUS:** Reusablity, **LRN:** Learnability
UND: Understandability, **PER**: Performance, **MOD:** Modularity

RQ11.3: What are the design smells that developers perceive as the most harmful?

Approach To assess the perceived severity of multi-language design smells, we asked the developers to rank the design smells based on their perceived level of harmfulness (from the most harmful to the less harmful) using a score from 15 to 1 (15 for the most harmful, 1 for the less harmful). We asked the developers to consider the impact of the design smells on software quality during the ranking. We used the Borda Count technique to rank the candidates *i.e.,* design smells as described in Section 11.2. By considering the number of votes associated to each design smell, the Borda count yields a consensus-based ranking instead of a majority-based one.

Findings Table 11.5 reports the aggregated results of the Borda Count. Table 11.5 also provides the median perceived severity associated to each smell type. We found that *Not Handling Exceptions* is perceived as the most harmful design smells with a score of 2261, closely followed by *Assuming Safe Return Value* with a score of 2137. We also report the median severity associated to those smell types. Both *Not Handling Exceptions* and *Assuming Safe Return Value*received the highest median severity value of 12. The design smell *Local References Abuse* is also reported to be harmful with a median of severity of 11, but with a Borda Count of 2063. The developers also considered the smell *Memory Management Mismatch* to be harmful with a median severity of nine and a Borda score of 2052. The design smells *Excessive Inter-language Communication* and *Too Much Clustering* were also perceived by the developers as harmful with a median severity of 11 and 10 respectively, and a Borda Count score of 2040 and 1876 respectively.

The design smells *Hard Coding Libraries*, *Not Using Relative Path*, *Unused Method Declaration*, and *Unused Parameters* were reported to be less harmful compared to the other smells with a median severity value of five for all of them. Their Borda Count scores are 746, 632, 588, and 438, respectively.

These ranking results could be explained by the definition and the nature of the design smells. It is expected to consider *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, and *Memory Management Mismatch* among the most harmful design smells. Indeed, these types of design smells could be directly related to the introduction of bugs. Some of these design smells were indeed related to bugs, *e.g., Conscrypt*: `"Fixed various memory leaks and Java 8 JNI Compatibility WARNING in native method: JNI call made without checking exceptions when required to from CallObjectMethod"`). Several studies in the literature also discussed the bugs and issues that could result from mishandling

exceptions and memory issues [11,82]. The developers reported that they perceive the design smells *Unused Method Declaration* and *Unused Parameters* as less harmful than the others. However, we believe that all the design smell types should be considered with caution, including those that are perceived to be less harmful. In fact, even if the nature of some design smells may not seem directly related to bug, still these design smells can increase the maintenance effort and even lead to bugs, *e.g.,* , `"There were a bunch of exceptions that are being thrown from JNI methods that aren't currently declared"`, and `"Fix latent bug in unused method"` present examples of commit messages extracted respectively from *Conscrypt* and *Pljava*.

Summary of findings (RQ11.3): The design smells perceived to be the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering*. The design smells perceived as less harmful are: *Unused Parameters*,*Unused Method Declaration*, *Not Using Relative Path*, and *Hard Coding Libraries*.

RQ11.4: Do developers plan to refactor multi-language design smells?

Approach One way to ensure the software quality is the identification and refactoring of design smells occurrences. Therefore, we aim to investigate whether developers would consider refactoring the design smell occurrences, and if the refactoring considerations may vary from one specific type of smells to the other. For each code snippet in which the developers reported a design or implementation problem, we also asked them if they would consider applying any refactoring to remove the identified problem. Specifically, we proposed a solution and asked them whether they would refactor with the given solution, refactor with an alternative solution, or whether they would not refactor. We then computed for each smell type, the percentage of developers who reported that they would refactor the code with the proposed solution, refactor the code with an alternative solution, or would not refactor the code. We also aggregate the responses to provide a general overview of the developers' opinion about refactoring multi-language design smells.

Findings Figure 11.3a and Figure 11.3b present the percentage of developers (for the open an closed surveys, respectively) who (1) selected the option No, when we asked them if they would consider refactoring the design smell presented in the code snippet; (2) selected "Yes" for the given solution; or (3) selected "Yes" for any alternative refactoring solution. We also report in Table 11.6 an overview of the developers responses for the combination of both

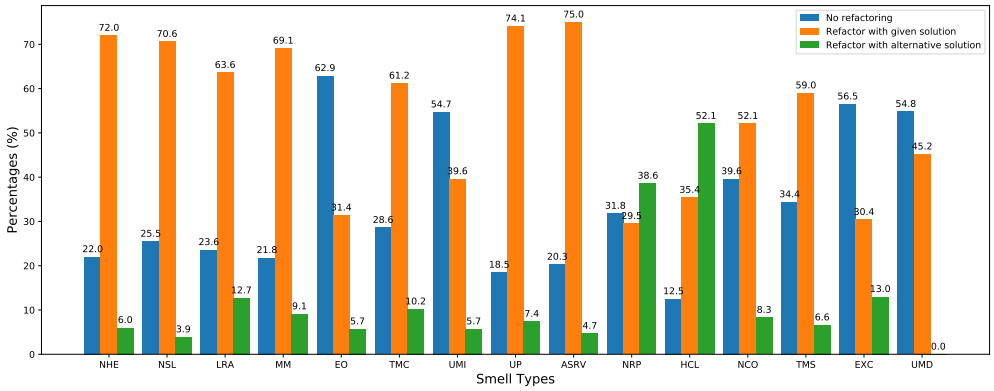
Table 11.5 Ranking of Multi-language Design Smells from Developers' Perception

Design Smells	Score (Borda)	Median Severity
Not Handling Exceptions	2261	12
Assuming Safe Return Value	2137	12
Local References Abuse	2063	11
Memory Management Mismatch	2052	9
Excessive Inter-language Communication	2040	11
Too Much Clustering	1876	10
Not Securing Libraries	1358	7
Too Much Scattering	1342	7
Excessive Objects	1211	6
Unused Method Implementation	964	5
Not Caching Objects	812	6
Hard Coding Libraries	746	5
Not Using Relative Path	632	5
Unused Method Declaration	588	5
Unused Parameters	438	5

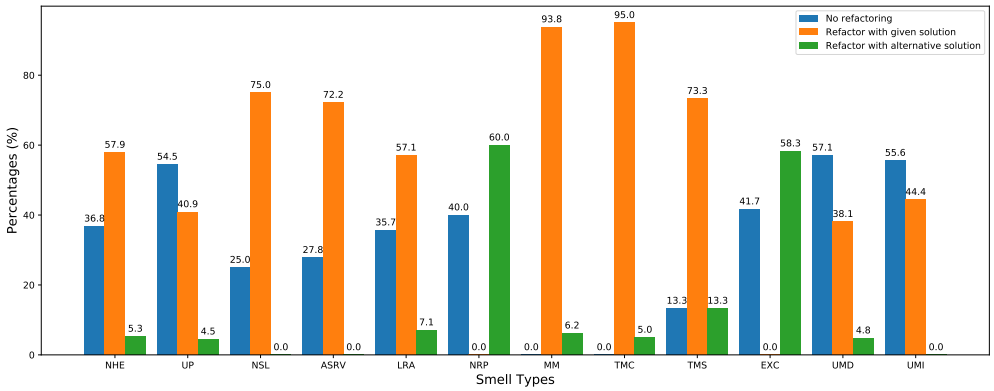
open and closed surveys. From our results in Table 11.6, we can observe that developers' decision on refactoring is dependent on the design smells types.

Smells that would be refactored

From the results presented in Table 11.6, one can see that for the design smell *Memory Management Mismatch*, 81.45% out of all the developers reported that they would apply the proposed refactored solution, 7.65% reported that they would consider another solution, while 10.9% reported that they would not consider refactoring that design smell. For the design smell *Too Much Clustering*, 78.1% of the developers reported that they would consider the proposed refactoring solution, 14.3% would not consider refactoring that design smell, while 7.6% would consider refactoring the smell using another solution. For the design smells *Assuming Safe Return Value* and *Not Securing Libraries*, 73.6% and 72.8% of the developers respectively reported that they would consider the proposed solution for refactoring, 24.05% and 25.25% of the developers respectively selected the option to not refactor, while 2.35% and 1.95% of the developers respectively reported that they would refactor the design smells with another solution. For the design smells *Not Handling Exceptions* and *Local References Abuse*, respectively 64.95% and 60.35% out of all the developers from the open and closed surveys reported that they would consider applying the proposed refactoring solution, 5.65% and 9.9% reported that they would refactor using an alternative solution, while 29.4% and 29.65% respectively reported that they do not consider refactoring the design smell. Table 11.6 also show that 66.15% and 57.5% of the developers respectively reported for the design smells *Too Much Scattering* and *Unused Parameters* that they would consider applying the



(a) General Developers



(b) Original Developers

Figure 11.3 Developers' Perceived Refactoring Considerations for the Design Smells

proposed refactoring solution, 23.85% and 36.5% respectively reported that they would not consider refactoring the design smells, while 9.95% and 5.95% respectively selected the option of refactoring with an another solution.

Smells that would not be refactored

When asking the developers if they would consider refactoring the occurrences of the design smell *Excessive Objects*, 62.9% of the developers selected the option "No", meaning that they would not refactor the code to remove the design smell, 31% reported that they would consider applying the given refactoring solution, while 5.7% reported that they would use an alternative refactoring solution to remove the smell. This result could be explained by safety concerns. When features are mixed together with excessive calls between components written in different programming languages, a change to the behavior of one may cause a bug in another feature. Therefore, developers may be reluctant to remove the design smell

Table 11.6 Overview Developers' Results For Refactoring the Smells (Open and Closed Surveys)

Smell	#No	#Yes_Given	#Yes_Alternative	%No	%Yes_Given	%Yes_Alternative
NHE	18	47	4	29.4	64.95	5.65
NSL	19	54	2	25.25	72.8	1.95
LRA	18	43	8	29.65	60.35	9.9
MMM	12	53	6	10.9	81.45	7.65
EO	22	11	2	62.9	31.4	5.7
TMC	14	49	6	14.3	78.1	7.6
UMI	34	25	3	55.15	42	2.85
UP	22	49	5	36.5	57.5	5.95
ASRV	18	61	3	24.05	73.6	2.35
NURP	16	13	20	35.9	14.75	49.3
HCL	6	17	25	12.5	35.4	52.1
NCO	19	25	4	39.6	52.1	8.3
TMS	23	47	6	23.85	66.15	9.95
EIC	31	14	13	49.1	15.2	35.65
UMD	46	36	1	55.95	41.65	2.4

to avoid possible side effects that can cause bugs. For the design smells *Unused Method Implementation* and *Unused Method Declaration*, 55.95% and 55.15% of the developers respectively reported that they would not consider refactoring the design smell, 41.65% and 42% respectively reported that they would apply the proposed refactoring solution, while 2.4% and 2.85% respectively selected the option of refactoring using an alternative solution.

From Figure 11.3a and Figure 11.3b, we observe that there are a few cases where the choice about the refactoring varies between the open and closed surveys. For example, for the design smell *Unused Parameters*, while in the open survey 75% of the developers reported that they would consider applying the proposed refactoring, 54.5% of the original developers (in the closed survey) reported that they would not consider refactoring that design smell. The same goes for the design smell *Excessive Inter-language Communication*; while 56.5% of the developers of the open survey reported that they do not consider refactoring the design smell, 58.3% of the original developers (in the closed survey) reported that they would consider refactoring the design smell with an alternative solution.

From Figure 11.3a and Figure 11.3b, we observe that in general, all the developers reported that they would consider refactoring the design smells. This may indicate that multi-language design smells are perceived by the developers as harmful, and refactoring them is likely to be among their priorities. This could be resulting from the impacts of the design smells on software quality. As reported in the two previous research questions, the developers perceive the design smells as harmful and consider them to have a negative impact on software qual-

ity. In addition, in most of the situations, the developers reported that they would consider applying the refactoring solutions that we proposed. Therefore, we believe that a refactoring approach could be considered to improve the quality of multi-language systems by removing the occurrences of multi-language design smells. From analyzing bug-fixing commit messages, we identified some commits reporting a refactoring for specific smells *e.g.,* `"removing unused parameter", "implementing the handling of exception"`. Those commit messages suggest that developers often refactor occurrences of the studied design smells.

> Summary of findings (RQ11.4): The refactoring consideration varies from one specific smell type to another. However, in general the developers reported that they would consider refactoring the design smells.

11.4 Discussion

We now discuss the results reported in Section 11.3.

11.4.1 Developers' Perception of Multi-language Design Smells

Relevance of Multi-language Design Smells From our results in **RQ11.1**, we observe that developers were mainly able to correctly identify the following design smells: *Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, *Memory Management Mismatch*, *Unused Parameters*, *Not Handling Exceptions*, and *Too Much Clustering*. The design smell *Unused Method Implementation* is defined by a native method that is implemented but never called from the host code, while the design smell *Unused Method Declaration* is about a native method that is declared in the host code but is never implemented in the native code. In the same vein, the design smell *Unused Parameters* is about a parameter that is passed from the java to the native code without being part of the native implementation. These design smells are related to unused code. Therefore, it is not very surprising that the developers were able to identify them easily. Since multi-language systems are emerging from the concept of combining components written in different languages and they generally involve different developers who might not be part of the same team. It could also be a challenging task for a developer working only on a sub-part of a project to clearly determine whether that specific parameter or method is used by other components.

Regarding the design smell *Too Much Clustering* it is also not surprising to see that developers were able to identify it correctly, because we believe that this type of design smell is widespread; developers may frequently observe occurrences of this design smell type [129].

The design smells *Memory Management Mismatch* and *Not Handling Exceptions* are commonly discussed in several research articles and developers' blogs discussed bugs related to mishandling JNI exceptions and the management of the memory [11, 49, 82]. Therefore, it is also not very surprising that most of the developers were able to identify these types of design smells. Similarly, another commit message was reporting on the design smells *Not Handling Exceptions* and *Memory Management Mismatch* in *Conscrypt* (`"rework exceptions throwing from jni", "Added error handling of all uses of sk_*_push which can fail due to out of memory"`). On the other hand, most of the developers were not able to correctly identify the design smells *Hard Coding Libraries*, *Excessive Objects*, *Not Using Relative Path*, and *Not Caching Objects*. From our results, most of the responses related to the identification of these design smell types are resulting from the open survey as shown in Table 11.2. Therefore, the perceived relevance for these design smell types is resulting from the general developers and not the original developers.

Impacts of Multi-language Design Smells on software quality From analyzing the results of **RQ11.3**, we found that the developers reported design smells *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering* as the most harmful types of design smell. While the design smells *Unused Parameters*,*Unused Method Declaration*, *Not Using Relative Path*, and *Hard Coding Libraries* are perceived as less harmful.

Considering the design smells *Not Handling Exceptions* and *Assuming Safe Return Value* as harmful could be explained by the possible issues that could arise from these types of design smells. Indeed, the management of exceptions may not be automatically ensured depending on the programming languages. In some situations, developers should explicitly implement the exception handling flow. Similarly, return values are often used in multi-language systems to pass objects and values from one language to the other. Thus, it is important to ensure that the interaction between language is successfully completed. Otherwise, bugs and issues may occur [10, 11, 49, 82]. The design smells *Local References Abuse* and *Memory Management Mismatch* were both reported as harmful. These design smells may lead to memory leaks which is also commonly discussed in the JNI development literature. The management of the memory should be considered separately for both Java and C/C++ sides. Unlike Java, the C language requires developers to explicitly take care of the management of the memory using available functions such as `malloc` and `free`. However, this leak and mismatch between Java and C/C++ is a source of programming defects since it is introducing security vulnerabilities. The manual process of managing the memory is likely to introduce memory leaks. Thus, developers should pay more attention to memory management when they are in the context of multi-language development. An

example of commit message in *Realm* reflects a case of bug related to the design smell *Local References Abuse* (`"Local ref needs to be cleaned on client thread (#4830), Clean the local ref after notifyAllChangesDownloaded"`). In *Rocksdb*, we also found a commit message describing a bug resulting from the design smell *Memory Management Mismatch* (`"As raised in #2265, the arena allocator will return memory that is improperly aligned to store a ‘std::function‘ on macOS."`)

Refactoring of Multi-language Design Smells From our results in **RQ11.4**, we found that in general, the developers would consider refactoring design smells occurrences. Our results also suggest that the developers would apply the proposed solution for refactoring. Hence, we believe that an approach to automatically remove such occurrences has a good chance of adoption by developers. During our analysis, we also observed situations where the original developers of a smelly file reported that they would not consider refactoring the occurrence of the design smells. This is the case for example of the design smell *Unused Parameters*. We attribute these developers' decisions to the risk of side effects that could result from refactoring parts of the code on which they have an imperfect knowledge. We also attribute these decisions in part to the nature of multi-language programming. Since multi-language systems may involve different teams who contribute separately using different programming languages, developers' may not have a global view of the whole system to decide whether for example a parameter could be removed without causing bugs or breaking changes to other related components (*e.g.,* in the case of the *Unused Parameters*design smell). However, as presented in some examples of extracted commit messages, there were situations in which the developers explicitly reported removing occurrences of the design smell *Unused Parameters* because there was bugs resulting from this design smell type. Therefore, we believe that developers should consider refactoring occurrences of multi-language design smells whenever possible.

11.4.2 Comparative Insights Regarding Previous Works

In Chapter 9, we studied the prevalence of design smells in open source projects. Our results show that most of the design smell types are prevalent in open source projects, in particular *Unused Parameters*, *Too Much Scattering*, *Unused Method Declaration*, while others are less prevalent, *e.g., Excessive Objects* and *Not Caching Objects*. The results from surveying developers confirm this finding about the prevalence of the design smells. Therefore, we believe that most of the design smells studied in this thesis are prevalent and are considered by developers to reflect design and implementation problems.

Regarding the severity of multi-language design smells, in the study presented in Chapter

9, we reported that files with occurrences of multi-language design smells are more subject to bugs than files without those types of design smells. We also report that some specific types of design smells are more related with bugs than others: *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding Libraries*, *Not Handling Exceptions*, *Memory Management Mismatch* and *Not Securing Libraries*. From surveying developers, we found that in general, most of the design smells have a negative impact on software quality attributes, and that they are considered as harmful by developers. However, some specific types are reported to be more harmful than the others: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering* are reported to be the most harmful design smell types. The design smells *Hard Coding Libraries*, *Not Using Relative Path*, *Unused Method Declaration*, and *Unused Parameters* were reported by the developers as less harmful than the other types of smells. Comparing the findings from our previous study and this study, we observe that while some types of design smells are considered by developers as less harmful, they are among of the most harmful design smells and were reported to increase the risk of the introducing of bugs as presented in Chapter 9. Indeed, developers reported that the design smells *Unused Parameters* and *Hard Coding Libraries* are less harmful compared to other types of design smells. However, from analyzing open source projects, we found that these two types of design smells are considered among the most bug-prone types of design smells. Both developers' perception and the empirical evidence presented in Chapter 9 and Chapter 10 are important regarding the impacts of the smells on the quality of multi-language systems. Although, both studies provide us with important insights about the impacts of multi-language design smells, we believe that further research is important to have more conclusive evidence.

11.5 Threats To Validity

In this section, we discuss the threats to the validity of our study [118].

Threats to Construct Validity These threats concern the relationship between theory and observation. Concerning the measure of perception, we asked developers to tell us whether they perceived a problem in the code shown to them. In addition, we asked them to specify the smell name in order to understand whether or not they were able to correctly identify the design smells studied. We are aware that surveys only reflect a subjective perception of the problem, and might not fully capture the extent to which the design smells could be prevalent or harmful. Regarding the sample of smells used in this survey, to mitigate any

possible threats related to the recall and-or precision of the detection approach, we manually validated all the occurrences of design smells that were used in the survey.

Threats to Internal Validity One of the main threats with a survey is that developers misunderstand the questions and–or possible answers. To minimise this threat, we relied on the literature to extract the possible questions and their answers (for close questions). We also relied on the literature to design the whole study [110]. We provided, in the survey preamble, the definition of all the terms used (*e.g.,* quality attributes, multi-language systems, and design smells). For most of the survey questions, we allowed developers to select a "null" option: "Other (please specify)" if they did not want to answer or wanted to answer differently. Our surveys received a higher response rate than the average response rate in software engineering surveys. This was because for the closed survey, we sent personalized emails, with the developer name and the project name. For the open survey, we also sent personalized message through LinkedIn. Such specific information increases the chance of contacted developers responding to our email compared to emails with only generic contents. Another reason is that we sent a reminder to developers after two weeks. We have limited a possible bias effect by also showing source code elements without smells to mitigate any bias that could be introduced by developers that could have reported that they perceived the presence of smells even in code not containing any smell.

Threats to External Validity We target software developers in general but also those who contributed in the smelly files. We targeted professionals from different countries and backgrounds. However, the results reflect the perception of specific developers and could not be generalized to all developers. Also, the impacts and relevance of smells reports the perception of the surveyed developers. Such perception depends on the smell instances and could vary from one participant to another. In this study, we had to constrain our analysis to a limited set of smell instances per survey. For the open survey, we were constrained to split the smells instances in two surveys because the number of questions and tasks to be performed by each respondent had to be reasonably small. For the closed survey, we also presented to each participant, the smells existing in files he worked on. This may introduce threats because not all the smells were presented to all the developers. However, we are aggregating the results and reporting if the presented smells were correctly identified by the developers. Thus, further investigations should be done before generalizing the results of the perceived relevance and severity of multi-language design smells.

Threats to Conclusion Validity We diversified the keywords and tried to reach developers with different backgrounds and skills. We included general developers but also those who were involved in the development and maintenance of the studied smells.

Threats to Reliability Validity We mitigate this threat by providing all the information needed to reproduce this study. We discussed in Section 11.2 the tool used as well as the methodology followed to perform this study. We also explained all the different steps followed to collect and analyze the data[7].

11.6 Chapter Summary

In this chapter, we presented the results of our survey of professional developers that aimed to access their perception of the relevance and severity of the multi-language design smells proposed in Chapter 7. We also reported about developers' perception of the impact of the design smells on some quality attributes. We selected respondents from different background to diversify the population under study. Our results show that (1) overall, developers consider the proposed design smell to be reflective of design and implementation problems. (2) The design smells are perceived in general to negatively impact all the studied quality attributes. (3) The design smells perceived as the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, and *Excessive Inter-language Communication*. (4) In general, the developers would consider refactoring the design smells from their systems.

[7] `https://github.com/ResearchML/RegisteredReport_Perception`

CHAPTER 12 CONCLUSION

In this chapter, we conclude the thesis and summarize our findings. In addition, we will discuss the limitations of our studies and the directions for future work.

12.1 Thesis Findings and Conclusions

Most complex software systems are developed using components written in different languages and technologies [3]. These systems provide a great variety of services to numerous different types of users. Developers use the programming languages best suited for their needs, to cope with challenges of building complex systems, instead of trying to implement all the functionalities with a single programming language. The benefits of reuse and advantages of each programming language are driving factors behind the use of multiple programming languages in the same project. However, these systems also pose significant challenges to developers. Multi-language systems are difficult to analyse, understand, and maintain [3, 5, 16]. Software quality has been widely studied in the literature and is often related to the occurrence of design smells. However, for multi-language systems, and despite their increasing popularity, such design smells are not yet well established. Therefore, in this thesis, we conducted quantitative and qualitative studies to define, catalog, and document design smells for multi-language systems. We also investigated their prevalence and impacts on software quality.

We formulated the following statement:

Thesis Statement:
Design smells (1) exist in multi-language systems; (2) they are prevalent in open source projects; and they (3) negatively impact the software quality.

In this dissertation, we have verified our thesis statement and proved that multi-language systems introduce new types of design smells, that those design smells are prevalent, and that they have negative impacts on the software quality, *e.g.,* fault-proneness.

12.2 Summary of the Findings

In the following paragraphs, we summarize our main contributions regarding the stated three sub-hypothesis. The following is organized regrading our objectives discussed in Chapter 4.

State of art and state of practice of Multi-language Systems

In this dissertation, we took a step back before addressing our three sub-hypothesis, and performed two pilot studies presented in Chapter 5 and Chapter 6, to capture the state of art of multi-language systems presented in the literature but also in practice, *i.e.,* in the industry. These two studies served as the base of our motivation to acquire necessary knowledge and materials to address our objectives presented in Chapter 4 and investigate our thesis sub-hypothesis.

We stared by conducting a systematic literature review (SLR) on multi-language systems. Our study presented in Chapter 5 covered 10 years (from 2010 to 2020). Our results show that the number of published papers discussing multi-language systems has been increasing over time with a sharp increase between 2017 and 2019. Most of the studied papers also point that multi-language development is continuously increasing thanks to the several benefits offered by such systems. We found that 66.66% of the studied papers belong to the "Software analysis" category. Several studies also discussed the challenges of multi-language systems, and 20.29% of the papers claimed that design patterns and design smells present the major concern for multi-language systems. Furthermore, we identified 60 combinations of programming languages discussed in the literature, while most papers focused on the analysis of the combination of Java and C/C++ (37.68%), through the Java Native Interface.

Besides surveying the literature, we also surveyed professional developers (in Chapter 6) to capture their perception about the benefits, challenges, and identify possible practices that they are using when developing multi-language systems. We surveyed a total of 131 professional developers. Our results support the findings from the literature. Indeed, the participants highlighted the growing popularity of multi-language development. They reported that multi-language systems introduce several benefits but that those systems also introduce challenges related to the complexity due to the nature of multi-language systems.

Sub-hypothesis 1: There exist design smells specific to multi-language systems

Our findings from the two previous studies highlighted that despite the huge interest in multi-language systems and their popularity, there is a lack of formal guidelines that developers could consider when dealing with multi-language systems. Therefore, with the aim of improving the quality of multi-language systems, we extracted, encoded, and cataloged good and bad practices in the development, maintenance, and evolution of multi-language systems, in the form of design smells. We performed a qualitative study and analysed the source code of open-source multi-language systems as well as developers' documentation, bug reports,

developers' blogs, and programming-language specifications. We presented in Chapter 7 a total of 15 types of multi-language design smells.

The detection of design smells can substantially reduce the cost of maintenance and development activities. Since in our thesis, we are documenting a new catalogue of multi-language design smells, there are no existing approach or tools to detect such occurrences. Therefore, in this thesis we proposed an approach to detect the occurrences of multi-language design smells. The approach currently supports design smell detection in the context of JNI systems. We evaluated the proposed approach with six open source projects and found a minimum precision and recall of 88% and 74%, respectively.

Based on these findings, we validated our first sub-hypothesis by showing that there exist design smells specific to multi-language systems. We reported 15 types of multi-language design smells.

Sub-hypothesis 2: Multi-language design smells are prevalent

Existing studies on mono-language systems, reported that occurrences of design smells may hinder the evolution of a system and increase the cost of maintenance activities. Therefore, we conducted an empirical study and investigated the prevalence of multi-language design smells in open source projects. Our results show that design smells are prevalent in the selected projects and persist across the releases. Some design smells are more prevalent than others, *e.g., Unused Parameters*, *Too Much Scattering*, *Unused Method Declaration* while others are less prevalent, *e.g., Excessive Objects* and *Not Caching Objects*.

In addition to the empirical investigation of the prevalence of multi-language design smells, we also conducted surveys with general developers and with original developers that contributed to the smelly files. We proposed a set of tasks containing occurrences of multi-language design smells and asked the developers whether they considered that the code snippet represent any design or implementation issues. Our results show that most of the design smells were correctly identified by developers. We also found that some specific types of smells were identified more frequently than others.

Our findings validate our second sub-hypothesis. We have shown through empirical studies that the studied multi-language design smells are prevalent in open source projects.

Sub-hypothesis 3: Design smells have a negative impact on software quality

Given the known impacts of design smells on the quality of mono-language systems, it is also important to study the impacts of multi-language design smells on software quality.

Therefore we conducted two empirical studies combining quantitative and qualitative analysis to investigate the impacts of multi-language design smells on software fault-proneness. We also surveyed developers about their perceived impacts of multi-language design smells.

The results of this dissertation show that multi-language design smells impact software quality negatively. From the empirical study conducted in Chapter 9, we found that files with multi-language smells are more likely to be subject of bugs than files without those smells. We also found that some specific smells, are more likely to be of a concern than others, *i.e., Unused Parameters*, *Too Much Scattering*, *Too Much Clustering*, *Hard Coding Libraries*, and *Not Handling Exceptions*. These smells seem more related to faults, than others. Hence, we suggest that practitioners consider them in priority for testing and-or refactoring.

When investigating the survival time of multi-language smelly files in Chapter 10, we found that multi-language smelly files experience bugs faster than files without those smells and that files without multi-language smells have hazard rates 87.5% lower than files with multi-language smells. Also, multi-language smells are not equally bug-prone. Developers should consider giving a special attention to files containing *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, and *Unused Method Declaration* design smells.

To gain better insights into the impacts of multi-language smells on software fault-proneness, we performed two qualitative analysis using both manual analysis and automated topic modeling as detailed in Chapter 9 and Chapter 10, to extract respectively, the activities that are more likely to introduce bugs, and categories of bugs related to multi-language smells.

We report in Chapter 9, that the activities related to data conversion, memory management, code restructuring, API usage, and exception management are the most common activities that could increase the risk of bugs once performed in smelly files, and thus should be performed carefully. We report in Chapter 10, that programming errors, libraries and features support, and memory issues are the most dominant types of bugs in multi-language smelly files.

From surveying developers, we found that design smells are perceived as having a negative impact on software quality attributes. Most of the quality attributes were reported to be negatively impacted by the design smells. The design smells perceived as the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, and *Excessive Inter-language Communication*. Developers' decision to refactor or not the design smells varies based on the type of the design smells. However, in general the participants to our survey reported that they would consider refactoring the design smells.

The above findings confirmed the third sub-hypothesis. We have shown that as for mono-language design smells, multi-language design smells also impact software quality negatively.

12.3 Implication of the Findings

Based on our results we formulate some recommendations and highlight the implications of our findings that could help the researchers, the developers, the academia, and also anyone considering using more than one programming language in a software system:

Our main objectives in this dissertations were: (1) to document and catalogue multi-language design smells, (2) investigate their prevalence, (3) and study their impacts on software quality. Our results report that there are design smells specific to multi-language systems, that they frequently occur within the studied projects and that they may increase the risk of bugs occurrence. Our results also highlight that the frequency and impact differ from one smell to the other. We also found that files with multi-language design smells experience bugs faster than files without those smells, and that some specific types of design smells are more bug-prone compared to other design smells types considering the time to the introduction of a bug. Our results also report that while some specific types of design smells were not perceived by developers as harmful, they have a negative impact on the software fault-proneness. Therefore, we believe that developers and researchers should pay attention to files containing the studied multi-language design smells. We also studied the activities that could introduce bugs once performed in smelly files. Those activities should be considered with caution while performed in smelly files as they were reported to introduce bugs. Also, our identified categories of bugs in multi-language systems could help developers and testers set maintenance priorities for those smells. For example, our results show that general programming errors constitute the most dominant category of bugs, and that design smells *Unused Method Declaration*, and *Excessive Inter-language Communication* are the most frequently occurring design smells related to that category of bugs. Therefore, prioritizing these types of design smells for refactoring could reduce possible bugs and consequently improve the quality of multi-language systems.

To Researchers- More research is needed to define design patterns and design smells for multi-language systems. Design smells in mono-language systems have been widely studied in the literature and have been found to impact program comprehension [104] and increase the risk of bugs [24]. However, the impact of multi-language smells on software quality is still under-investigated. Researchers could find interest in studying why and how some specific types of smells are more frequent than others and the reasons behind their increase over time. They could also investigate the reasons why some specific types of smells are

more related to bugs than others. Our findings also suggest that files with multi-language smells experience bugs faster than files without those smells. Thus, to help improve the quality of multi-language systems, we encourage researchers to deeply study such systems, analyze design patterns and design smells, and empirically evaluate their impacts on software quality. As we observed that multi-language design smells are related to bugs and that they contribute to accelerate the introduction of bugs, researchers could also explore the causes and circumstances under which the studied smells may increase the risk of bugs. They could also investigate the roots causes and recommend mitigation strategies related to the categories of bugs that could result from the occurrences of multi-language design smells. The same goes for the activities, they could investigate further reasons behind the introduction of bugs when those specific activities are performed. They could also explore the existence of other activities that could introduce bugs.

Additionally, the catalogue of design smells studied in this thesis is not exhaustive and presents only a small subset of possible multi-language smells and practices. Therefore, researchers and developers could further investigate smells and practices in multi-language software development.

To Developers- As our results show that multi-language design smells are related to bugs, developers should consider removing such smells. Developers could take advantage of the outcome of this thesis to reduce the maintenance cost of multi-language systems. In fact, most of the smells discussed in this thesis (even those that are not always related to bugs) could introduce additional challenges and increase the effort of maintenance activities. Having knowledge of their existence and potential impact could help to improve the quality of multi-language systems, and avoid their introduction in systems during maintenance activities. In fact, as reported in Chapter 9 and Chapter 10, we found multiple commit messages in which developers explicitly mentioned issues caused by the occurrence of a smell studied in this thesis. Studying each type of smell separately also allowed us to capture their impact individually. The insights from this study could help developers to prioritize multi-language smells for maintenance and refactoring activities. We believe that the smell types *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, *Too Much Scattering*, *Too Much Clustering*, *Hard Coding Libraries*, and *Unused Method Declaration* should be considered in priority since their occurrence seems to increase the risk of bug introduction. The same goes for the categories of bugs. The developers could leverage our results to better prioritize bugs. They could also leverage our results to better prioritize their refactoring activities. The same goes for the activities introducing bugs. Being aware of those activities could help developers avoid issues when performing them.

To Academia- The multi-language development has brought several advantages to software engineering. However, to better benefit from multi-language development, formal guidelines should be considered. Since our results highlight the importance and impacts of multi-language design smells on software bug-proneness, we believe that providing courses discussing the multi-language systems could help to support the quality assurance of multi-language systems. The academic community can also explore example case studies to teach future developers about good practices to adopt and bad practices to avoid when developing multi-language systems.

12.4 Limitations

- This dissertation reports on multi-language design smells and their impacts on software quality. However, in our experimentation we focused on one specific type of multi-language systems, *i.e.,* JNI systems. While, the studied design smells were extracted from resources (*e.g.,* open source systems, developers' documentation, bug reports) that include not only JNI systems but also other combination of systems, the detection approach presented in Chapter 8 as well as the empirical studies presented in Chapter 9, Chapter 10, Chapter 11 considered only JNI systems. Therefore, the results could vary if we consider other combination of programming languages.
- Our thesis is an internal validation of a subset of multi-language design smells. Hence, our results are related to the design smell types studied in this thesis, other types of design smells may lead to other results. Therefore, replicating the studies presented in this thesis with other types of design smells could add more insights about the impacts of multi-language design smells on software quality.
- A part of our conclusions relies on manual analysis or surveys. Future replications with larger sample size or larger number of survey participants could be considered to increase the generalizability of our conclusions.

12.5 Opportunities for Future Research

This dissertation is a first step towards a comprehensive study of multi-language design smells and their impacts on software quality. We have verified our thesis and proved that by identifying and tracking multi-language design smells, it is possible to improve the quality of multi-language systems. The maintenance and evolution of multi-language systems, design patterns and design smells for multi-language systems, and the quality assurance of multi-language systems are topics that are currently under-explored by the research community.

Our thesis findings open a wide range of opportunities for future work. In the future, we plan to extend our study in the following directions:

- Most of our case studies through this dissertation focus on the combination of Java and C/C++, through the Java Native Interface. This choice was motivated by findings from the systematic literature review presented in Chapter 5. However, today we observe a large volume of modern software systems being developed with other combinations of programming languages. Therefore, analyzing these systems could lead to the definition of new design patterns for multi-language systems.
- In this dissertation, we reported on multi-language design smells. In the future, we plan to investigate design patterns for multi-language systems. Defining and documenting design patterns could considerably reduce the challenges related to multi-language systems.
- From our findings in Chapter 11, we reported that developers would consider refactoring design smells. This was also confirmed by the empirical studies from Chapter 9 and Chapter 10, where we capture commit fixing messages in which developers' explicitly reported refactoring some of the design smells studied in this thesis. Our results from surveying developers in Chapter 6, show that the perceived effort to remove bad practices is considered as a challenging task. Therefore, we believe that refactoring strategies could also be considered to support the quality of multi-language systems.
- One important aspect that we plan to investigate in our future research work is the co-occurrence of multi-language design smells with traditional smells (that can occur in components written in a single language).
- In the future, we will expand our study to other sets of programming languages. We also plan to extend the proposed smell detection approach to integrate other sets of languages.
- We plan to extract a taxonomy of multi-language bugs along with their root causes and formulate mitigation strategies.
- We plan to perform case studies with professional developers and students to investigate the impact of multi-language smells on software maintenance and evolution tasks.
- We also plan to study machine learning application developed with multiple programming languages. We will extract design smells and design patterns related to these application, and investigate the categories of bugs and issues encountered by developers when implementing a multi-language machine learning application.

REFERENCES

[1] P. K. Linos, Z.-h. Chen, S. Berrier, and B. O'Rourke, "A tool for understanding multi-language program dependencies," in *Program Comprehension, 2003. 11th IEEE International Workshop on.* IEEE, 2003, pp. 64–72.

[2] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on.* IEEE, 2006, pp. 497–500.

[3] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 563–573.

[4] B. Kullbach, A. Winter, P. Dahm, and J. Ebert, "Program comprehension in multi-language systems," in *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on.* IEEE, 1998, pp. 135–143.

[5] P. Mayer and A. Schroeder, "Cross-language code analysis and refactoring," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on.* IEEE, 2012, pp. 94–103.

[6] B. D. Burow, "Mixed language programming," in *Computing in High Energy Physics' 95: CHEP'95.* World Scientific, 1996, pp. 610–614.

[7] T. C. Jones, *Estimating software costs.* McGraw-Hill, Inc., 1998.

[8] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 3, p. 12, 2009.

[9] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 62–72.

[10] M. Abidi, M. Grichi, and F. Khomh, "Behind the scenes: developers' perception of multi-language practices," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering.* IBM Corp., 2019, pp. 72–81.

[11] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 365–377.

[12] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, "Debug all your code: Portable mixed-environment debugging," *SIGPLAN Not.*, vol. 44, no. 10, pp. 207–226, Oct. 2009.

[13] M. Goedicke, G. Neumann, and U. Zdun, "Object system layer," *5th European Conference on Pattern Languages of Programms (EuroPLoP '2000)*, 2000.

[14] M. Goedicke and U. Zdun, "Piecemeal legacy migrating with an architectural pattern language: A case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 1, pp. 1–30, 2002.

[15] A. Neitsch, K. Wong, and M. W. Godfrey, "Build system issues in multilanguage software," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 140–149.

[16] Z. Mushtaq and G. Rasool, "Multilingual source code analysis: State of the art and challenges," in *Open Source Systems & Technologies (ICOSST), 2015 International Conference on*. IEEE, 2015, pp. 170–175.

[17] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[18] R.-H. Pfeiffer and A. Wąsowski, "Texmo: A multi-language development environment," in *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ser. ECMFA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 178–193.

[19] Z. Mushtaq and G. Rasool, "Multilingual source code analysis: State of the art and challenges," in *2015 International Conference on Open Source Systems Technologies (ICOSST)*, Dec 2015, pp. 170–175.

[20] F. Boughanmi, "Multi-language and heterogeneously-licensed software analysis," in *2010 17th Working Conference on Reverse Engineering*, Oct 2010, pp. 293–296.

[21] T. Arbuckle, "Measuring multi-language software evolution: A case study," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the*

7th Annual ERCIM Workshop on Software Evolution, ser. IWPSE-EVOL '11. New York, NY, USA: ACM, 2011, pp. 91–95.

[22] H. P. Langtangen, "A case study in high-performance mixed-language programming," in *International Workshop on Applied Parallel Computing*. Springer, 2006, pp. 36–49.

[23] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 75–84.

[24] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 294–305.

[25] F. Khomh, "Patterns and quality of object-oriented software systems," Ph.D. dissertation, University of Montreal, Québec, Canada, 2010. [Online]. Available: http://hdl.handle.net/1866/4601

[26] C. Alexander, S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, and I. Fiksdahl-King, *A pattern language*. Gustavo Gili, 1977.

[27] 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.

[28] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[29] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[30] ——, "Do code smells reflect important maintainability aspects?" in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 306–315.

[31] 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.

[32] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 437–446.

[33] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 393–402.

[34] M. Abidi, F. Khomh, and Y.-G. Guéhéneuc, "Anti-patterns for multi-language systems," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*. ACM, 2019, p. 42.

[35] F. Tomassetti and M. Torchiano, "An empirical assessment of polyglot-ism in github," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 17:1–17:4.

[36] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering, 2007. FOSE '07*, 2007.

[37] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno, "Towards the detection of cross-language source code reuse," in *Proceedings of the 16th International Conference on Natural Language Processing and Information Systems*. Springer-Verlag, 2011.

[38] M. Harman, "Why source code analysis and manipulation will always be important," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, Sept 2010, pp. 7–19.

[39] H. M. Kienle, J. Kraft, and H. A. Müller, "Software reverse engineering in the domain of complex embedded systems," in *Reverse Engineering-Recent Advances and Applications*. InTech, 2012.

[40] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '03. IBM Press, 2003, pp. 266–278.

[41] S. Liang, *Java Native Interface: Programmer's Guide and Reference*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[42] J. Hunt, *Java for Practitioners: An Introduction and Reference to Java and Object Orientation*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[44] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[45] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[46] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[47] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[48] B. F. Webster, *Pitfalls of object-oriented development*. M & T Books, 1995.

[49] M. Abidi, M. Grichi, F. Khomh, and Y.-G. Guéhéneuc, "Code smells for multi-language systems," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*. ACM, 2019, p. 12.

[50] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020.

[51] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information and Software Technology*, vol. 52, no. 9, pp. 902–922, 2010.

[52] M. Cleves, W. Gould, W. W. Gould, R. Gutierrez, and Y. Marchenko, *An introduction to survival analysis using Stata*. Stata press, 2008.

[53] S. Habchi, R. Rouvoy, and N. Moha, "On the survival of android code smells in the wild," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2019, pp. 87–98.

[54] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[55] S. W. Thomas, "Mining software repositories using topic models," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1138–1139.

[56] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 95–104.

[57] F. Gurcan and N. E. Cagiltay, "Big data software engineering: Analysis of knowledge domains and skill sets using lda-based topic modeling," *IEEE Access*, vol. 7, pp. 82 541–82 552, 2019.

[58] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.

[59] M. Habibi and A. Popescu-Belis, "Keyword extraction and clustering for document recommendation in conversations," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 23, no. 4, pp. 746–759, 2015.

[60] P. K. Linos, "Polycare: A tool for re-engineering multi-language program integrations," in *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95*. IEEE, 1995, pp. 338–341.

[61] S. Li and G. Tan, "Finding bugs in exceptional situations of jni programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 442–452.

[62] ——, "Jet: Exception checking in the java native interface," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. ACM, 2011, pp. 345–358.

[63] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 109–118.

[64] J. Ebert, B. Kullbach, V. Riediger, and A. Winter, "Gupro-generic understanding of programs an overview," *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, pp. 47–56, 2002.

[65] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars." in *CASCON*, vol. 3. Citeseer, 2003, pp. 266–278.

[66] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, "Traceback: first fault diagnosis by reconstruction of distributed control flow," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 201–212.

[67] D. Strein, H. Kratz, and W. Lowe, "Cross-language program analysis and refactoring," in *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2006, pp. 207–216.

[68] P. Linos, W. Lucas, S. Myers, and E. Maier, "A metrics tool for multi-language software," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. Citeseer, 2007, pp. 324–329.

[69] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 303–312.

[70] L. Gong *et al.*, "Java security architecture (jdk 1.2)," *Draft Document, revision 0.8, Sun Microsystems, March*, 1998.

[71] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, 2017.

[72] D. L. Moise and K. Wong, "Extracting and representing cross-language dependencies in diverse software systems," in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, 2005, pp. 10–pp.

[73] H.-C. Fjeldberg, "Polyglot programming," Ph.D. dissertation, Master thesis, Norwegian https://www.overleaf.com/project/5c799d12e492137c3d36a21bUniversity of Science and Technology, Trondheim/Norway, 2008.

[74] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *TAPOS*, vol. 2, no. 1, pp. 3–13, 1996.

[75] J. K.-Y. Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioural and creational design motifs through dynamic analysis," *Journal of Software: Evolution and Process*, vol. 22, no. 8, pp. 597–627, 2010.

[76] H. Kampffmeyer, S. Zschaler, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, "Finding the pattern you need: The design pattern intent ontology," in *MoDELS*, vol. 4735. Springer, 2007, pp. 211–225.

[77] Y.-G. Guéhéneuc, J.-Y. Guyomarc'h, and H. Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study," *Software Quality Journal*, vol. 18, no. 1, pp. 145–174, 2010.

[78] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, pp. 667–684, 2008.

[79] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.

[80] C. Alexander, *The timeless way of building*. New York: Oxford University Press, 1979, vol. 1.

[81] M. Goedicke, G. Neumann, and U. Zdun, "Message redirector," *6th European Conference on Pattern Languages of Programms (EuroPLoP '2001)*, 2001.

[82] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang, "Safe Java Native Interface," in *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006, pp. 97–106.

[83] A. Malinova, "Design approaches to wrapping native legacy codes," *Scientific works, Plovdiv University*, vol. 36, pp. 89–100, 2008.

[84] A. Osmani, *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. " O'Reilly Media, Inc.", 2012.

[85] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 97–106.

[86] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[87] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[88] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic generation of detection algorithms for design defects," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 297–300.

[89] N. Moha and Y.-G. Guéhéneuc, "P tidej and d ecor: identification of design patterns and design defects," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 868–869.

[90] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[91] ——, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[92] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[93] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 81–90.

[94] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 113–122.

[95] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[96] G. Rasool and Z. Arshad, "A lightweight approach for detection of code smells," *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 483–506, 2017.

[97] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[98] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 385–396.

[99] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "Deep learning anti-patterns from code metrics history," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 114–124.

[100] ——, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, p. 110486, 2020.

[101] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[102] A. Borrelli, V. Nardone, G. A. Di Lucca, G. Canfora, and M. Di Penta, "Detecting video game-specific bad smells in unity projects," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 198–208.

[103] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.

[104] 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.

[105] C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y.-G. Guéhéneuc, and A. Maiga, "A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension," *Information and Software Technology*, vol. 122, p. 106278, 2020.

[106] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 232–243.

[107] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of sql code smells in data-intensive systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 327–338.

[108] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[109] ——, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[110] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[111] Z. A. Kermansaravi, M. S. Rahman, F. Khomh, F. Jaafar, and Y.-G. Guéhéneuc, "Investigating design anti-pattern and design pattern mutations and their change-and fault-proneness," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–47, 2021.

[112] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[113] M. Staples and M. Niazi, "Experiences using systematic review guidelines," *J. Syst. Softw.*, vol. 80, no. 9, pp. 1425–1437, Sep. 2007.

[114] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A systematic literature review on the usage of eye-tracking in software engineering," *Inf. Softw. Technol.*, pp. 79–107, Nov. 2015.

[115] R. Bluff, "9 rosalind bluff grounded theory : the methodology," in *9 ROSALIND BLUFF Grounded theory : the methodology*, 2005.

[116] A. Fink, *The survey handbook*. Sage, 2003, vol. 1.

[117] A. Strauss and J. Corbin, *Basics of qualitative research techniques*. Sage publications Thousand Oaks, CA, 1998.

[118] R. K. Yin, *Applications of Case Study Research Second Edition (Applied Social Research Methods Series Volume 34)*. {Sage Publications, Inc}, 2002.

[119] G. Neumann and U. Zdun, "Pattern-based design and implementation of an xml and rdf parser and interpreter: A case study," in *European Conference on Object-Oriented Programming*. Springer, 2002, pp. 392–414.

[120] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, and F. Khomh, "State of practices of java native interface," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 274–283.

[121] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[122] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[123] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[124] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *Java coding guidelines: 75 recommendations for reliable and secure programs*. Addison-Wesley, 2013.

[125] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 516–519.

[126] G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing xpath queries," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 444–491, 2005.

[127] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[128] S. Liang, *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[129] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, "Are multi-language design smells fault-prone? an empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–56, 2021.

[130] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 908–911.

[131] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[132] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases." in *icsm*, 2000, pp. 120–130.

[133] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 677–687.

[134] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2017.

[135] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. Chapman and Hall/CRC, 2003.

[136] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[137] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Software Engineering*, vol. 13, no. 5, p. 473, 2008.

[138] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 13–21.

[139] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo, "Cataloging github repositories," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2017, pp. 314–319.

[140] D. Blei, L. Carin, and D. Dunson, "Probabilistic topic models: A focus on graphical model design and applications to document and image analysis," *IEEE signal processing magazine*, vol. 27, no. 6, p. 55, 2010.

[141] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 189–198.

[142] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[143] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 351–360.

[144] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.

[145] M. Castelluccio, L. An, and F. Khomh, "An empirical study of patch uplift in rapid release development pipelines," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3008–3044, 2019.

[146] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. González-Barahona, "What if a bug has a different origin? making sense of bugs without an explicit bug introducing change," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–4.

[147] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

[148] E. C. Neto, D. A. d. Costa, and U. Kulesza, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–12.

[149] C. Treude and M. Wagner, "Predicting good configurations for github and stack overflow topic models," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 84–95.

[150] H. Jelodar, Y. Wang, C. Yuan, X. Feng, X. Jiang, Y. Li, and L. Zhao, "Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey," *Multimedia Tools and Applications*, vol. 78, no. 11, pp. 15 169–15 211, 2019.

[151] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in javascript projects," *Software Quality Journal*, vol. 27, no. 3, pp. 1271–1314, 2019.

[152] Q. Till, "How to ship product with a quarterly product roadmap and sprint-based execution," in *website*, 2019. [Online]. Available: https://www.getshipit.com/blog/how-to-ship-product-with-a-quarterly-product-roadmap/

[153] P. Thongtanunam and A. E. Hassan, "Review dynamics and their impact on software quality," *IEEE Transactions on Software Engineering*, 2020.

[154] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 2019, pp. 2–11.

[155] A. Radu and S. Nadi, "A dataset of non-functional bugs," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 399–403.

[156] R. Lima, J. Souza, B. Fonseca, L. Teixeira, R. Gheyi, M. Ribeiro, A. Garcia, and R. de Mello, "Understanding and detecting harmful code," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 223–232.

[157] C.-M. Tan, Y.-F. Wang, and C.-D. Lee, "The use of bigrams to enhance text categorization," *Information processing & management*, vol. 38, no. 4, pp. 529–546, 2002.

[158] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008, pp. 304–318.

[159] F. Gravetter, "Forzano, lab research methods for the behavioral sciences," 2012.

[160] S. Baltes and S. Diehl, "Worse than spam: Issues in sampling software developers," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–6.

[161] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 274–278.