Empirical Software engineering (EMSE) manuscript No.
(will be inserted by the editor)

Design Smells in Multi-language Systems and Bug-proneness: A Survival Analysis

Mouna Abidi · Md Saidur Rahman ·
Moses Openja · Foutse Khomh

Accepted: March 11, 2024

Abstract Modern applications are often developed using a combination of programming languages and technologies. Multi-language systems offer opportunities for code reuse and the possibility to leverage the strengths of multiple programming languages. However, multi-language development may also impede code comprehension and increase maintenance overhead. As a result of this, developers may introduce design smells by making poor design and implementation choices. Studies on mono-language systems suggest that design smells may increase the risk of bugs and negatively impact software quality. However, the impacts of multi-language smells on software quality are still under-investigated. In this paper, we aim to examine the impacts of multi-language smells on software quality, bug-proneness in particular. We performed survival analysis comparing the time until a bug occurrence in files with and without multi-language design smells. To have qualitative insights into the impacts of multi-language design smells on software bug-proneness, we performed topic modeling and manual investigations, to capture the categories and characteristics of bugs that frequently occur in files with multi-language smells. Our investigation shows that (1) files with multi-language smells experience bugs faster than files without those smells, and non-smelly files have hazard

Mouna Abidi
DGIGL, Polytechnique Montreal
E-mail: mouna.abidi@polymtl.ca

Md Saidur Rahman
DGIGL, Polytechnique Montreal
E-mail: saidur.rahman@polymtl.caa

Moses Openja
DGIGL, Polytechnique Montreal
E-mail: moses.openja@polymtl.ca

Foutse Khomh
DGIGL, Polytechnique Montreal
E-mail: foutse.khomh@polymtl.ca

rates 87.5% lower than files with smells, (2) files with some specific types of smells experience bugs faster than the other smells, and (3) bugs related to "programming errors", "external libraries and features support issues", and "memory issues" are the most dominant types of bugs that occur in files with multi-language smells. Through this study, we aim to raise the awareness of developers about the impacts of multi-language design smells, and help them prioritize maintenance activities.

Keywords Design Smells, Code Smells, Multi-language Systems, Mining Software Repositories, Empirical Studies, Survival Analysis

1 Introduction

Design smells have been defined by Fowler as symptoms of poor design and implementation choices that can have adverse impacts on software quality [17]. 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. Design smells in mono-language systems have been widely studied in the literature and have been found to impact program comprehension [1] and increase the risk of bugs [66]. Several studies report that classes containing design smells are significantly more bug-prone and change-prone than classes without smells [30,31,64,69,81].

However, existing studies on design smells primarily focus on mono-language projects and do not consider the smells related to the interaction between components written in different programming languages (*i.e.,* multi-language design smells). Multi-language development offers several advantages to software engineering [33,37]. Today, developers frequently use more than one programming language to develop a single application. Developers no longer need to reinvent the wheel by implementing the code from scratch. Instead, they can reuse existing components and external libraries to reduce the development time and cost [2,37,73]. By combining programming languages, multi-language systems also allow overcoming the weaknesses that could be related to some programming languages. The possibility to leverage the strengths of multiple programming languages and to reuse existing code samples are two main 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 [33].

Multi-language systems usually rely on the 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. In this paper, we study one specific type of FFI, the JNI systems (*i.e.,* systems with Java and C/C++ programming languages). The advantages of multi-language systems come with new challenges. 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, studies report that multi-language development increases the cognitive overhead of code [33, 52, 63]. Such difficulties unfortunately might lead to bugs that are hard to detect and debug. An example of bugs related to the design smells discussed in this paper was reported in 2018 in `libguests`, due to a misuse of the JNI guidelines. There were missing checks for Java exceptions after JNI calls and also leak local references[1].

Since multi-language systems consist of combining programming languages with different semantics and lexical programming rules, those systems introduce new design patterns and design smells that consider the interaction between programming languages [3, 4, 20, 21, 22, 54]. There exist a few works in the literature that discussed the patterns, bad practices, and common issues related to multi-language programming [20, 21, 22, 54, 72, 73]. However, there has been limited focus on the impacts of multi-language smells on software quality. An extensive catalog of multi-language design smells was published by Abidi *et al.* [3, 4]. In a recent study, we performed an empirical investigation of the impact of the proposed catalog on software bug-proneness [5]. We analyzed 98 releases of nine open source multi-language projects. From our analysis, we found that the risk of bugs is higher in files with multi-language design smells than in files without occurrences of multi-language smells. We also found that some specific smells are more likely to introduce bugs than other design smells. However, in that study, we investigated the correlation between smells and bugs without considering the time-to-bug occurrence. Thus, to complement our previous work, in this paper, 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 prioritize the testing activities by knowing which smells should be considered in priority. Earlier studies investigating the lifespan and impact of design smells on software bug-proneness in mono-language systems also used survival analysis [24, 28, 51, 66]. Survival analysis is well-suited for capturing and analyzing time-to-event data, making it an appropriate choice for studying the time to bug occurrence in multi-language files.

In this study, we analyzed 270 snapshots of eight multi-language projects and performed survival analysis to compare the time until a bug occurrence in files with and without multi-language smells. The objective here is to investigate whether the files with multi-language design smells have lower survival probabilities to bug occurrence compared to files without those smells and to assess the impacts of multi-language smells on bug-proneness. To gain a deep insight into these impacts, we combine quantitative and qualitative analysis. In particular, we examine (1) the survival time of files with and without multi-language design smells until the occurrence of bugs, (2) the survival time of the files containing each type of multi-language design smell until the occur-

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

rence of bugs, (3) the categories of bugs that occur in files with multi-language smells, and (4) the design smell types associated with each bug category.

Our results show that: files containing multi-language smells experience bugs faster than files without a multi-language smell. Multi-language smells are not equally bug-prone. Developers should give particular attention to files containing design smells of types *Not Handling Exceptions*, *Local References Abuse*, *Memory Management Mismatch*, *Assuming Safe Return Value*, *Unused Parameters*, and *Unused Method Declaration* and prioritise them for refactoring. Programming errors, and issues related to libraries and features support, and memory are the most dominant types of bugs occurring in multi-language smelly files. 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. We believe that our results could help researchers and developers involved in the development of multi-language systems. Having knowledge of the potential impacts of multi-language smells could help to improve the quality of multi-language systems and reduce the challenges related to their maintenance and evolution. Developers could take benefits from knowing the design smell types that are more likely to introduce bugs to treat them as a priority and prevent their occurrences in the system.

The remainder of this paper is organized as follows. Section 2 presents related work and introduces background information about multi-language systems and presents the studied design smells. Section 3 describes our methodology. Section 4 reports our results, while Section 5 discusses these results and their implications. Section 6 summarises the threats to the validity of our work. Section 7 concludes the paper and discusses future works.

2 Background Information and Related Work

We present in this section the background of this study and discuss the literature related to this work.

2.1 Background Information

We provide in the following an overview of multi-language systems, Java Native Interface, and the studied design smells.

2.1.1 Multi-language Systems

Multi-language systems are systems that are developed with a combination of at least two programming languages. Such systems are gaining popularity because of their different inherent benefits. Developers often leverage the strengths of programming languages and reuse existing code to cope with the challenges of building complex systems [53, 59, 77]. However these systems also introduce new challenges related to their development and maintenance [33, 52].

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[26,41]. It allows to perform hardware and platform-specific features. JNI also increases the performance with the help of low-level libraries for computation and graphic operations [73].[2] JNI is widely studied in the literature, several studies discussed the benefits, challenges, and issues related to JNI systems. We present in Fig. 1 an example of a JNI code extracted from [41]. Fig. 1(a) presents a Java class that contains a native method declaration `Print()` and loads the corresponding native library while Fig. 1(b) presents the C file that contains the implementation of the native function `Print()`.

(a) JNI method declaration

```
class HelloWorld {

 static {
       System.loadLibrary("HelloWorld");}

    private native void print();

    public static void main(String[] args)
    { new HelloWorld().print();
    }
 }
```

(b) JNI implementation function

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv
     *env, jobject obj)
{
   printf("Hello World\n");
   return;
}
```

Fig. 1: JNI HelloWorld Example

2.1.2 Design Smells

Patterns were initially introduced in the domain of architecture by Alexander [7]. Gamma *et al.* [18] later introduced Design patterns in software engineering. Design patterns are defined as common guidelines and repeatable "good" solutions to solve recurrent problems. By contrast, design smells (*i.e.,* anti-patterns and code smells), are considered as symptoms of poor design and implementation choices. They represent violations of the best practices that often indicate the presence of bigger design or implementation problems [12,17]. In this paper we use design smells to refer to both code smells and anti-patterns. Several studies in the literature investigated the impacts of mono-language design smells (smells that occur in components written in a single programming language) and reported that files containing occurrences of design smells are significantly more bug-prone and change-prone compared to files without smells [30,64,69,81].

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

Multi-language design smells are defined as poor design and implementation choices that affect the quality of multi-language systems. A few papers in the literature discussed the design patterns and design smells related to multi-language systems [20,21,22,54,72,73]. However, an extensive catalog was published by Abidi *et al.* [3,4]. We briefly describe in Table 1 the design smells studied in this paper extracted from the catalog proposed by Abidi *et al.* [3,4]. While some of the studied design smells (*e.g.,* Not Handling Exceptions, Not Securing Libraries, Too Much Clustering) could apply to the context of mono-language systems, in this study we are considering exclusively the situations where the design smells occur in the context of multi-language systems by considering the interaction of programming languages. For example, for the design smell **Not Handling Exceptions**, we consider the exceptions resulting from the combination of programming languages. In the case of JNI, when the native code is being called from Java, the exceptions do not disturb the control flow. Handling of such exceptions is postponed until returning back to Java code. Therefore, developers should explicitly implement the exception handling mechanism for any exception that occurs in the native code. Mishandling exceptions, especially in multi-language environments, can lead to security vulnerabilities [72,73]. For the design smell **Not Securing Libraries**, we consider situations where the native library is not implemented in a secured block to ensure that the library cannot be loaded without permission. For the design smell **Too Much Clustering**, we consider situations where the class contains an excessive number of native methods that are declared in that class but implemented in the native code. As outlined in the previously published catalog [3,4], multi-language systems inherently pose greater difficulties in comprehension and introduce additional challenges compared to mono-language systems. These challenges primarily stem from the incompatibilities among programming languages and the heterogeneity of system components. The presence of design smells occurrences in such systems is anticipated to amplify the maintenance-related challenges and introduce heightened complexity. Identifying occurrences of design smells and-or fixing bugs across various programming languages could be a challenging task for maintainers.

Figure 2 illustrates an example of JNI code with occurrences of some of the studied smells. In Fig. 2(a), *System.loadLibrary("Vulnerable")* presents an occurrence of the smell *Not Securing Libraries*. The native library is loaded without any security checks. Thus, malicious code can call native methods from the library, this may impact the security and reliability of the system [3, 47]. A possible solution would be to load the library within a secured block via *AccessController.doPrivileged*. In Fig. 2(b), the use of data returned by native methods *e.g.,* GetByteArrayElements without performing checks or throwing exceptions presents occurrence of the smell *Not Handling Exceptions* and could lead to bugs and leave security breaches open to malicious code [34,40].

Listing 1 presents an example extracted from *Rocksdb*. In this example, the method `GetIntArrayElement` is used to capture a Java array. However, the memory is not released using ReleaseObjectArrayElement.

(a) Design Smell - Not Securing Libraries

```
private native void bcopy(byte[] arr);
public void byteCopy(byte[] arr) {
bcopy(arr);}
static{  System.loadLibrary("Vulnerable");    }
```

(b) Design Smell - Not Handling Exception

```
void Java_Vulnerable_bcopy
(JNIEnv *env, jobject obj, jbyteArray jarr) {
char buffer[512];
if  ((*env)->GetArrayLength(env, jarr)    > 512) {
JNU_ThrowArrayIndexOutOfBoundsException(env,0);}
jbyte *carr=(*env)
GetByteArrayElements(env,jarr,NULL);
strcpy(buffer, carr);}
```

Fig. 2: Example of Multi-language Smelly Code

Listing 1: Design Smell - Memory Management Mismatch

```
/* 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);;
```

Listing 2 illustrates an example of the smell *Assuming Safe Return Value* extracted from *OpenDDS*. Here, if the class `clazz` or one of its methods is not found, the native code will cause a crash as the return value is not checked properly. Checking return values in the context of multi-language code allows confirming that the call to a method from one programming language to another programming language was performed correctly. We present in Listing 3 an example of the design smell *Local References Abuse*, where local references are created but not deleted accordingly.

Listing 2: Design Smell - Assuming Safe Multi-language Return Values

```
/* C */
CORBA::Object_ptr ptr = recoverTaoObject(jni, jThis);
  if (CORBA::is_nil(ptr)) return 0;
  CORBA::Object_ptr dupl = CORBA::Object::_duplicate(ptr);
  jclass clazz = jni->GetObjectClass(jThis);
  jmethodID ctor = jni->GetMethodID(clazz, "<init>", "(J)V");
  return jni->NewObject(clazz, ctor, reinterpret_cast<jlong>(dupl));
}
```

Table 1: List of the Studied Multi-language Design Smells

Design Smells	Definition
Assuming Safe Return Value	Not checking multi-language return values may lead to errors and security issues.
Excessive Inter-language Communication	A wrong partitioning in components written in different languages leads to many calls in one way or the other.
Excessive Objects	Passing excessive objects from Java to native code could lead to extra overhead to properly handle Java types.
Hard Coding Libraries	Not correctly loading library could bring confusion and make it hard to know which library has really been loaded.
Local References Abuse	Pay attention to the number of references created and always deleted the local references once not needed.
Memory Management Mismatch	JNI handles Java objects as reference types by allocating memory. That memory should be released after usage.
Not Caching Objects	Looking up class objects from native code is expensive, it is recommended to globally cache commonly used IDs.
Not Handling Exceptions	Mishandling native exceptions may lead to vulnerabilities and leave security breaches open to malicious code.
Not Securing Libraries	A common way to load the native library in JNI is the use of the method *loadLibrary* without a secure block.
Not Using Relative Path	Library is loaded by using an absolute path to the library instead of the corresponding relative path.
Too Much Clustering	Too many native methods declared in a single class would decrease readability and maintainability of the code.
Too Much Scattering	Classes are scarcely used in multi-language communication without satisfying both the coupling and the cohesion.
Unused Method Implementation	A method declared in the host language and implemented in the native code. However, this method is never used.
Unused Method Declaration	A method is declared in the host language, however is never implemented in the native code.
Unused Parameters	Parameters are present in the method signature however they are no longer used in the other programming language.

Listing 3: Design Smell - Local References Abuse

```
/* C++ */
 for (int i = 1; i < len; ++i) {
     jstring arg = reinterpret_cast<jstring>
                 (env->GetObjectArrayElement(args, i - 1));    }
```

2.2 Related Work

We discuss in the following studies relevant to our work.

2.2.1 Multi-language Systems:

Several studies in the literature discussed multi-language systems [29,35,38, 45,46,54]. Kullbach *et al.* [37] investigated the quality of multi-language systems. They reported 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. Linos *et al.* [46] later argued that no attention has been paid to the issue of measuring multi-language systems' impact on program comprehension and maintenance. They proposed *Multi-language Tool (MT)*; a tool for understanding and managing multi-language programming dependencies. Kontogiannis *et al.* [35] 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 argued that creating dedicated multi-language systems, methods, and tools to support such systems is expected to have an impact on the software maintenance process which is not yet known. Similarly, Kochhar *et al.* [33] studied open-source projects from GitHub to investigate 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 claimed that design patterns and anti-patterns were present in multi-language systems and suggested that researchers study them thoroughly. They also suggested further studies to investigate the benefit of using multiple languages in developing software systems. Kondoh *et al.* [34] presented four types of JNI mistakes made by developers. They proposed a static analysis tool to find a set of common mistakes in critical code sections. Tan *et al.* [73] studied a range of bug patterns in the native code of the JDK and extracted six common bugs. The authors proposed an approach to prevent these bugs. Li and Tan [40] reported the risks induced by the exception mechanisms in Java. They also highlighted the bugs that could result in the native code from mishandling exceptions.

2.2.2 Impacts of Design Smells:

Several studies have investigated the impact of smells on software quality [15, 32, 44, 55, 56, 57, 58, 60, 69, 79, 80]. However, most of these studies target mono-language systems. Khomh *et al.* [30] reported that classes with occurrences of design smells are more likely to be the subject of changes than classes without those occurrences. Olbrich *et al.* [55] proposed an approach that analyses 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.* [1] conducted three experiments to capture the impact of occurrences of design smells on developers' understandability of systems when performing maintenance tasks. This study has been replicated by Politowski *et al.* [60], confirming the findings. They reported that the occurrence of one type of smell does not significantly impacts program comprehension. However, the combination of two types of smells negatively impacts program comprehension. Saboury *et al.* [66] 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. Muse *et al.* [51] 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. Morales *et al.* [50] performed an empirical study on three open source projects investigating the impacts of code review practices on software design quality. They considered seven types of mono-language design smells. Their results suggest that software components with low review coverage are often more prone to the occurrences of design smells compared to software components with more active review coverage. They reported that even if code review coverage has an impact on the occurrence of design smells, it is not sufficient to avoid the occurrences of design smells. They argued that good code review practices could help improving the software design quality. Borrelli *et al.* [11] 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. Abidi *et al.* [6] performed an empirical study on 98 releases of JNI systems and investigated prevalence and impacts of multi-language design smells on software quality. They reported that multi-language smells are prevalent in open-source projects and that they persist throughout the releases of the systems. They also claimed that some kinds of smells are more prevalent than others. Their results suggest that multi-language smells can often be more associated with bugs than files without smells. Our study is complementary to this previous work. However, our analysis is at a finer level of granularity (*i.e.,* snapshots level). Furthermore, in the previous study we report the correlation between smells and bugs without considering the time to bug occurrence. In this study we emphasize on the timeline and risk level of the introduction of bugs. Correlation and survival analysis are both useful to study the impact associated with design smells. Therefore, we believe that these two studies are complementary and important to clearly capture the impact of multi-language design smells on software bug-proneness. In addition, we are reporting the most dominant types of bugs that occur in multi-language smelly files.

In contrast with the studies discussed in this section, our work presents the first empirical study that investigates the impact of multi-language smells on the time of bug occurrences. We believe that our study could serve as a guideline for further studies on multi-language design patterns and smells and their impacts on software quality.

3 Study Design

We present in this section our methodology to perform this study. Figure 3 provides an overview of the methodology.

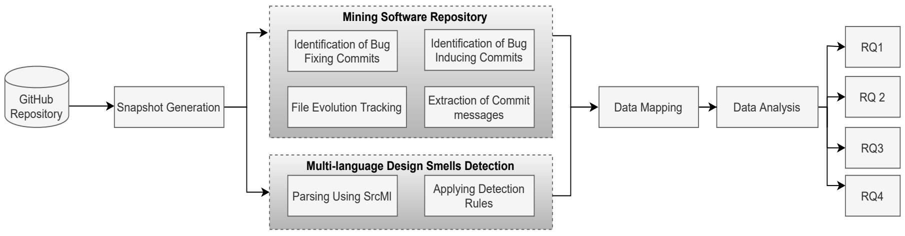

GitHub Repository
Snapshot Generation
Mining Software Repository
Identification of Bug Fixing Commits
Identification of Bug Inducing Commits
File Evolution Tracking
Extraction of Commit messages
Multi-language Design Smells Detection
Parsing Using SrcMl
Applying Detection Rules
Data Mapping
Data Analysis
RQ1
RQ 2
RQ3
RQ4

Fig. 3: Schematic Diagram of the Study

3.1 Setting Objectives of the Study

Our goal in this study is to investigate the relationship between the occurrence of multi-language design smells and the occurrence of bugs over time. 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 mitigating the impacts of multi-language design smells on software bug-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 bug-proneness of in-house multi-language projects. This study considers 15 types of multi-language design smells presented in the catalog of multi-language design smells [4,3]. 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:

- **RQ1: 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 bug-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.
- **RQ2: 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.

– **RQ3: What are the categories of bugs that exist in multi-language smelly files?** To better capture the impacts of multi-language smells on software bug-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.
– **RQ4: 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.

3.2 Data Collection and Data Extraction

3.2.1 ***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 2 summarizes the characteristics of the selected systems.

Among the selected systems, *JNA*[3] is a native shared library. It provides Java programs easy access to native shared libraries. *Rocksdb*[4] is developed and maintained by Facebook. It presents a persistent key-value store for fast data storage, it can also be used for client-server database. *Javacpp*[5] provides efficient access to native C++ inside Java, unlike some C/C++ compilers that interact with assembly language. *Realm*[6] is a mobile database that runs directly inside tablets and smartphones. *Pljava*[7] is a free module that uses the standard JDBC interface to bring Java Stored Procedures and Functions to

[3] `https://github.com/java-native-access/jna`
[4] `https://github.com/facebook/rocksdb/`
[5] `https://github.com/bytedeco/javacpp`
[6] `https://github.com/realm/realm-java`
[7] `https://github.com/tada/pljava`

Table 2: Overview of the Studied Systems

Projects	Domain	#Snapshots	LOC	Java	C/C++
Conscrypt	Cryptography (Google)	32	91,765	85.3%	14%
Frostwire	File and Media Sharing	18	403,106	71.4%	19%
Javacpp	Compiler	30	28,713	98%	0.6%
JNA	Native Shared Library	32	590,208	70.2%	15.4%
OpenDDS	Adaptive Communication	58	2,803,495	5%	16%
Pljava	Database	35	71,910	67%	29.7%
Realm	Mobile Database	29	171,705	82%	8.1%
Rocksdb	Facebook Database	36	487,853	11%	83.1%

the PostgreSQL backend. *Frostwire*[8] is a file sharing client and media management tool. *Conscrypt*[9] is a Java Security Provider (JSP) that is developed and maintained by Google. It implements parts of the Java Cryptography Extension (JCE) and Java Secure Socket Extension (JSSE). *OpenDDS*[10] is an open source C++ implementation of the Object Management Group (OMG) Data Distribution Service (DDS).

3.2.2 ***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 [76]. We selected a total of 270 snapshots from the eight subject systems (shown in Table 2).

3.2.3 ***Identification of Bug-Fixing and Bug-Inducing Commits***

We use Github APIs and PyDriller to mine the software repositories and get the list of all the commit logs for our subject systems [70]. PyDriller offers a set of APIs that allows us to retrieve repository information including commits logs. 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*) using a heuristic similar to that presented by Mockus and Votta [49]. These keywords were used in multiple previous studies [5, 8, 49, 51]. 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 previous studies,

[8] `https://github.com/frostwire/frostwire`
[9] `https://github.com/google/conscrypt`
[10] `https://github.com/objectcomputing/OpenDDS`

we used PyDriller to capture the bug-inducing commits for each of the bug-fixing commits [5,39,42,51,62,75]. PyDriller leverages the SZZ algorithm to detect changes that introduced bugs (*i.e.,* earlier changes to the same code updated by bug-fixing commits) and returns bug-inducing commits for a given bug-fixing commit. To locate the bug-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. It returns the set of commits that previously changed the same lines of code using the git diff and blame commands. After extracting bug-inducing commits for our eight subject systems using PyDriller, we performed a manual inspection of the data prior to our analysis. The first and second author manually inspected the changes in all the bug-inducing commits reported for the system *Pljava*. Specifically, they proceeded as follows: for each bug-fixing commit, the two authors manually checked if the changes in the bug-inducing commits reported by Pydriller were indeed related to the corresponding bug-fixing commits. Through this analysis, they found the precision of the detection of bug-inducing commits to be 70.83%.

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.

3.2.4 ***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 [28]. 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.

3.2.5 ***Design Smells Detection***

We relied on the multi-language smell detection approach we proposed in our previous work [5]. The detection approach supports 15 types of multi-language smells as described in the recently published catalog of multi-language design smells [3,4]. The detection approach is based on a set of predefined detection rules extracted from the definition and documentation of the design smells. Those rules were validated during the documentation of the smells by the pattern community and are detailed in our previous work [3,4,5]. The approach relies on srcML[11], a parsing tool that converts source code into srcML, which is an XML format representation. The srcML representation adds into the source code text syntactic information as XML elements. We present in Listing 5 an example of the srcML representation of the code snippet presented in Listing 4. 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. The approach is applicable in the context of JNI. Thus, our study is based on JNI systems. The approach was evaluated with six JNI projects and was reported to have a minimum precision and recall of 88% and 74% respectively. We detect 15 types of multi-language smells on 270 snapshot of eight selected multi-language systems. Further details about the smell detection approach and its validation are presented in our previous work [5]. All the detection results are also available in our replication package.[12]

Listing 4: Example of Java Code

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

3.2.6 ***Mapping Bug-Inducing Commits with Design Smells***

Since our goal is to investigate the impact of multi-language design smells on software bug-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 $\mathrm{S}_{t+\alpha}$ 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.

3.3 Data Analysis

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

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

[12] https://github.com/ResearchMLS/Survival_Analysis

Listing 5: 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="HelloJNI.java"><class><specifier>
public</specifier> class <name>HelloJNI</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>
```

3.3.1 ***Survival Analysis***

Survival analysis are commonly used in medical research. Recently, researchers have been applying survival analysis to problems in Software Engineering [14, 24,66,67]. 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 [14,16, 24,43,66,67]. 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 (in our case the introduction of a bug). The rate is defined as a hazard rate. More specifically, Cox Hazard model analyze how long specific subjects can survive before a well-defined event occurs. The following function presents the hazard of the event of bug occurrence at a time t in Cox models:

$$\lambda_i(t) = \lambda_0(t) * e^{\beta * F_i(t)} \tag{1}$$

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

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_1 * f_{i1}(t) + ... + \beta_n * f_{in}(t) \tag{2}$$

Where,

– $F_i(t)$ 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 the function F$_i$(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 (bug occurrence in our case) 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:

$$\lambda_i(t)/\lambda_j(t) = e^{\beta*(f_i(t)-f_j(t))} \tag{3}$$

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.

Hazard ratios (HR) are measures of association widely used in prospective studies. It is the result of comparing the hazard function among exposed group (*i.e.,* smelly files in our case) to the hazard function among non-exposed group (*i.e.,* non-smelly files). Hazard ratio can be considered as an estimation of possible risk, which is the risk of an event occurrence (*i.e.,* bug occurrence in our case). 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.

To answer **RQ1**, we performed survival analysis and compared the time until the occurrence of bug in smelly and non-smelly files. 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 [28,66].

To answer **RQ2** we perform survival analysis for each type of smells, comparing the time until occurrence of a bug in files containing a specific smell and files without that smell. We follow the same approach as in **RQ1** 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 and 0 if it does not contain any smell of that type. Because the following metrics are known to be related to bug-proneness [36,66,68], 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.

3.3.2 ***Topic Modeling***

To answer **RQ3** and 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 removed stop words using MALLET [48] 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) [61]. We used the Gibbs [19] algorithms for Latent Dirichlet Allocation LDA [10]. LDA is a widely adopted topic modeling technique which is able to extract topics from smaller documents (*e.g.,* commit messages in our case). LDA was used in several studies in the literature [9,23,25,65,74]. LDA generates topics based on a set of frequently co-occurring keywords. It 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. Similar to previous study performing topic modeling approach, during the classification we rely on unigram and bi-gram to improve the quality of the text analysis [71]. 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 allowing us to gain insights about the types, characteristics, and causes of bugs related to multi-language smells. We randomly selected top 20 to 30 documents for each topic as performed in previous studies [71]. The topic files were shared between the first and second authors of this paper. The topics were assigned after reaching an agreement. We also group the bug topics into general categories of bugs that we report in Section 4.

To answer **(RQ4)**, we investigate the types of smells related to each bug category. In particular, here we reuse the categories and bug types resulting

Table 3: Bug Hazard Ratios for Each Project

Projects	**exp(coef)**	**p-value (CHM)**	**p-value (PHA)**
Conscrypt	2.598	3.218e-23	0.0001
Frostwire	3.123	1.749e-52	0.641
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
Pljava	1.805	6.425e-05	0.002
Realm	2.747	7.487e-37	9.112e-05
Rocksdb	1.64	6.162e-26	1.258e-05

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

from the previous research question **(RQ3)** and study the distribution of the different types of multi-language design smells in the buggy files.

4 Study Results

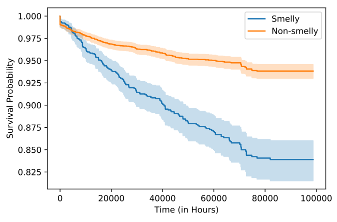
In this section we report our findings and answer each of the four research questions.

4.1 **RQ1: Is the risk of bugs higher in files with multi-language smells in comparison with those without smells?**

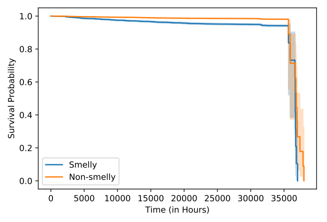
To assess the impacts of multi-language design smells on software bug-proneness, we performed survival analysis and compared the time until the occurrence of bug in smelly and non-smelly files.

Figure 4 provides the survival curves for native multi-language files with and without smells for all eight studied systems. Results presented in Fig. 4 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. 4 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 3 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.

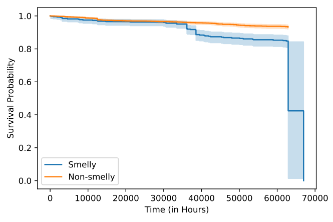
We believe that the results of hazard ratios computed for OpenDDS could be related to the type of design smells existing in this system, which motivated



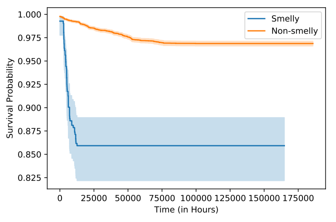
(a) Conscrypt



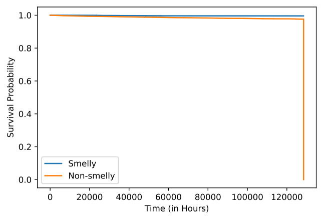
(b) Frostwire



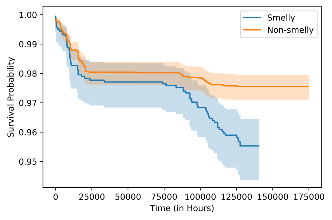
(c) Javacpp



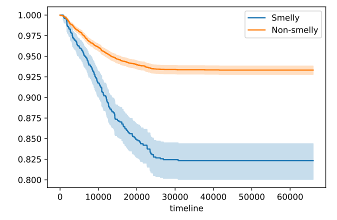
(d) Jna



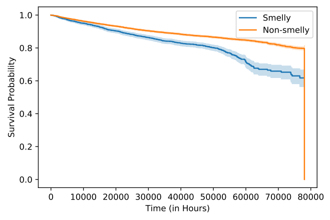
(e) OpenDDS



(f) Pljava



(g) Realm-java



(h) Rocksdb

Fig. 4: Survival Curves for Bug-occurrences in Files with (Smelly) and without (Non-smelly) Multi-language Smells

us to investigate in **RQ2** the survival until a bug occurrence for each type of smell. Our results for **RQ1** 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

Table 4: Summary of the Comparative Bug-proneness of Different Types of Multi-Language Smells

Smells	**#System**	**SFB**	**NSFB**	**% SFB**	**% NSFB**
ASRV	6	4	2	66.67%	33.33%
EO	0	N/A	N/A	N/A	N/A
EXC	7	5	2	71.43%	28.57%
HCL	2	2	0	<u>100.0%</u>	0.0%
LRA	6	5	1	<u>83.33%</u>	16.67%
MM	5	5	0	<u>100.0%</u>	0.0%
NCO	0	N/A	N/A	N/A	N/A
NHE	7	6	1	<u>85.71%</u>	14.29%
NRP	6	3	3	50.0%	50.0%
NSL	8	6	2	75.0%	25.0%
TMC	8	5	3	62.50%	37.50%
TMS	6	3	3	50.0%	50.0%
UMD	8	5	3	62.50%	37.50%
UMI	5	4	1	80.0%	20.0%
UP	8	7	1	<u>87.50%</u>	12.50%

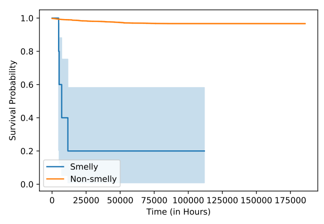
SFB: Number of Systems where smelly files are more bug-prone than non-smelly files
NSFB: Number of Systems where files without (specific) smells are more bug-prone than smelly files
#**System**: Number of Systems where we have hazard ratios for the concerned smell (covariate)
* Underlined percentage values indicate the top-5 bug-prone smells

(bugs occur faster) in files with multi-language design smells compared to files without those smells.

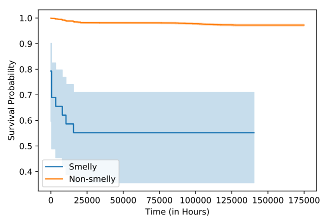
4.2 **RQ2: Is the risk of bugs equal from one multi-language design smell type to the other?**

Given the observed impacts of the multi-language smells on bug-proneness in RQ1, it is important to investigate the impact of each type of multi-language design smells. We performed survival analysis for each type of smells, comparing the time until occurrence of a bug as described in Section 3.

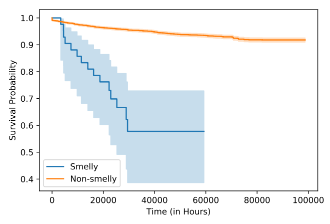
We report in Fig. 5 examples of the survival curves for individual smell types *Memory Management Mismatch* for *JNA*, *Local References Abuse* for *Pljava*, and *Not Handling Exceptions* and *Too Much Clustering* for *Conscrypt*. The complete results and figures are available in our replication folder.[12] 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 5 and Table 6, we present the bug hazard ratios for 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)



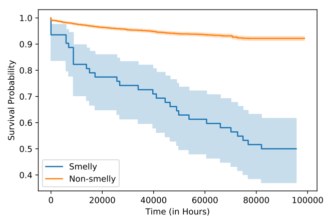
(a) Jna-Memory



(b) PlJava-References



(c) Conscrypt-Exception



(d) Conscrypt-Cluster

Fig. 5: Survival Curve for Native Smelly Files and Native Non Smelly Files for all the Projects

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 5 and Table 6. So, to evaluate the bug-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 5 and Table 6, 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 bug-proneness of individual types of multi-language smells, we present a summary of the results presented in Table 5 and Table 6 in Table 4. Here, each row in Table 4 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

Table 5: 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)
	LOC	2.59	–	3.218e-23	0.0001
	EXC	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
Conscrypt	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
	NRP	2.825	7	0.0065	4.411e-06
	MM	19.60	1	1.924e-27	0.0002
	LRA	19.077	2	1.167e-37	0.0001
	EXC	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
Frostwire	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
	NRP	0.403	–	0.365	0.144
	EXC	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
Javacpp	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
	PrevBugs	5.033	–	9.526e-32	1.254e-14
	EXC	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
JNA	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
	NRP	62.422	5	1.893e-16	0.533
	MM	60.852	6	2.919e-16	0.613
	LRA	33.298	8	1.343e-17	0.163

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 6: 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)
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
	NRP	6.139e-06	–	0.978	0.999
	LRA	6.139e-06	–	0.9787	0.999
Pljava	LOC	1.80	–	6.43e-05	0.0027
	EXC	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
	NRP	1.669e-05	–	0.991	0.99
	MM	2.613	–	0.176	0.877
	LRA	26.196	1	5.31e-30	1.857e-05
Realm	PrevBugs	2.746	–	7.487e-37	9.1123e-05
	EXC	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
	MM	6.88	2	8.748e-09	0.21
	LRA	6.975	1	0.0001	0.0020
Rocksdb	LOC	1.64	–	6.162e-26	1.258e-05
	EXC	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
	NRP	2.062	–	0.05	0.039
	MM	3.279	2	0.0003	0.087
	LRA	5.10	1	2.677e-07	0.939

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

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 5 and Table 6). 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 underlined percentage values in Table 4. 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 bug-proneness.

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 5 and Table 6) 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 [28, 66]. 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 bug-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.

4.3 **RQ3: What are the categories of bugs that exist in multi-language smelly files?**

To gain better insights into the impacts of multi-language smells on software bug-proneness, 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. 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.

As explained in Section 3.3.2, 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 bug categories based on the bug similarity perceived from the bug-fixing commit messages. We extend the bug categorization proposed by Ray *et al.* [63] to include bugs related to multi-language smelly files. Table 8 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 3.3.2, 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.

Table 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%

The assigned topics names and categories is inspired from the categorization proposed by Ray *et al.* [63] and also driven from rationale of the bug-fixing commit messages and the keywords. For example for Memory issues, the frequently co-occurring keywords represent activities and concepts related to memory management, such as memory table operations, compaction, size considerations, write and read operations, etc. For the performance issues, the reported keywords capture aspects related to speed, improvement, and general performance-related issues. For Concurrency, these keywords highlight elements related synchronization techniques, and thread-related concepts. For Libraries and Features Support, these keywords cover terms associated with updating, adding features, and making library calls.

Similar to previous work [33,63], we found that *programming errors* is the largest category of bugs related to multi-language smelly files (40.42%) as shown in Table 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 paper *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%) [72,73]. Such type of bugs are obvious as programming languages with unmanaged memory type allocations are known for bugs related to memory management [33]. In JNI development, when converting 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 the JNI development in network operations in some of the studied projects (*e.g., Conscrypt*, *Frostwire* and *OpenDDS*). [13]

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.[14] (`‘‘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.*

4.4 **RQ4: What are the dominant categories of bugs related to each type of multi-language smell?**

To complement the results of the previous research question (RQ3) 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 and study the occurrences and distribution of individual types of multi-language design smells. 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.

Table 9 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 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 also frequently occurs in the category of bugs related to general programming errors (19.4%) and errors related to the platform and dependencies (19.05%).

[13] `https://www.ibm.com/developerworks/library/j-transparentaccel/index.html`

[14] `https://medium.com/swlh/introduction-to-java-native\`
`\-interface-establishing-a-bridge-between-java-and-c-c-1cc16d95426a`

Table 8: Categories of Bugs and their Distribution in the Dataset

No	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 multi-threading	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 Updates	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

Similarly, the design smells *Unused Parameters* and *Unused Method Declaration* occur respectively with the 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 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. 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 9, 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 6 presents an example of bug-fixing commit in a smelly file extracted from *Rocksdb* (options.cc) presented in Section 2, Listing 1.[15] The `'+'` signs in this code example show 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 Section 2, JNI treats Java objects and

[15] `https://github.com/facebook/rocksdb/commit/c6d464a9da7291e776b5a017f0a5d33d61f2518b`

Listing 6: 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);
```

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 [3,73]. Occurrences of this design smell may also lead to memory leaks as presented in Listing 6. 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.

5 Discussion and Implications

This section discusses the results reported in Section 4.

5.1 Survival of Files with Multi-language Smells from Bugs:

From our results presented for **RQ1**, we observe that multi-language smells have a 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 bug-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 [30,66].

Table 9: Distribution of Multi-language Design Smells Among the Bug Categories

Bug Categories↓/Smells→	UP	UM	TMS	TMC	UMI	ASR	EO	EILC	NHE	NCO	NSL	HCD	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
UMI: UnusedMethodImplementation , **ASR:** AssumingSafeReturnValue, **EO:** ExcessiveObjects **EILC:**excessiveInterlangCommunication
NHE: NotHandlingExceptions,**NCO:** NotCachingObjects, **NSL:** NotSecuringLibraries,**HCD:** HardCodingLibraries
NURP: NotUsingRelativePath, **MMM:** MemoryManagementMismatch, **LRA:** LocalReferencesAbuse

Existing studies investigated other types of smells that do not consider the combination and interaction of programming languages [15,32,44,55,56,57, 58,60,69,79,80]. Therefore, we believe that reporting the results of the impact of multi-language design smells is complementary to existing work and essential and beneficial to assess the effectiveness and benefits of multi-language systems. 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 **RQ2** show that some specific types of smells are more related to bugs than others. The design smell *Not Handling Exceptions* is found to have a higher impact on bug-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 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 exceptions 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* which 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 from the native code. When an exception is thrown in the native code, answering such an 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 Management 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 references 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.

5.2 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 paper are important. Our results in **RQ3** 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 paper, 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 [33,63]. 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.[16] 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 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`").[17]
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 **RQ4** 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

[16] `https://codingforsmarties.wordpress.com/2017/04/02/breaking-changes/`
[17] `https://www.databasedevelop.com/article/12233882/How+to+resolve+dll+dependency+with+external+library.`

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[18]. 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 paper 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 [5,38,72]. A bug related to this smell type was also reported 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 Section 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 **RQ3** 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 **RQ4**, 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 **RQ2**, we report

[18] `https://www.tutorialfor.com/blog-219186.htm`

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.

5.3 Comparative Insights Regarding Previous Findings

In a previous work [5], our results 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 [3, 73, 72]. 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 our previous study [5]. 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 our previous study [5].

5.4 Implication of the Findings

From our results, we highlight the implications of our findings and formulate some recommendations that could help the researchers, the developers, the academia, and also anyone involved in the development of multi-language systems.

Our main objective in this paper was to investigate the survival probability of files with and without multi-language design smells to evaluate the impacts of those smells on the bug-proneness. Our results show 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 smell types. Therefore, we believe that developers should pay attention to files containing multi-language design smells. Also, our identified categories of bugs in multi-language systems could help developers set maintenance priorities for those smells. For example, our results show that the general programming errors present 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. More research should also be conducted to empirically investigate the impact of patterns and design smells on the quality of multi-language systems. Design smells in mono-language systems have been widely studied in the literature and have been found to impact program comprehension [1] and increase the risk of bugs [66]. However, the impact of multi-language smells on software quality is still under-investigated. Our findings 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 they contribute to accelerate the introduction of bugs. Therefore, 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 recommended mitigation strategies related to the categories of bugs that could result from the occurrences of multi-language design smells.

To Developers- As our results show that multi-language design smells are related to bugs, developers should consider removing such smells. Studying each type of smell separately also allowed us to capture their impacts individually. The insights from this study could help developers to prioritize multi-language design 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*, and *Unused Method Declaration* should be considered in priority. The same goes for the categories of bugs. The developers could benefit from our results by considering the categories of bugs that are highly impacted by multi-language design smells to consider them in priority for refactoring. Developers should also take into account the types, frequency, and severity of bugs associated with (each category of) smells when prioritizing them for refactoring or other maintenance activities.

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 of multi-language systems. The academicians can also explore example case studies to focus on the good and bad practices to adopt that could help to improve the quality of multi-language systems.

6 Threats To Validity

In this section, we discuss potential threats to the validity of our study following guidelines for empirical studies [82].

Threats to construct validity concern the relation between the theory and the observation. We relied on the smell detection approach proposed in a previous study [5]. 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 [28,51,66]. 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 [8, 49]. This method may not capture all the commits related to bug-fixing if the commit messages were not containing any of those keywords. Nevertheless, Castelluccio *et al.* [13] reported that this technique can achieve a precision of 87.3% and a recall of 78.2%. Moreover, the method that we used for mining bug-fixing and bug-inducing commits was evaluated in previous studies [5,51]. Muse *et al.* [51] manually validated the bug-inducing commits obtained by applying SZZ and found them to be highly accurate; they randomly selected 50 commits from the reported results and found only three false positives (6%). In this study, we have also manually inspected the changes in all the bug-inducing commits reported for *Pljava* and found the precision of our detection of bug-inducing commits to be 70.83%. 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 the 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 [27,78]. 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. Our investigation is an internal validation of previously defined and cataloged multi-language design smells [3, 4]. Therefore, the subset of multi-language design smells that we are considering may present a threat to validity. However, to mitigate this threat, we published our catalog in a pattern conference. The paper underwent multiple rounds of a shepherding process, where an expert in patterns and smells provided three rounds of meaningful comments to refine and enhance the patterns. Subsequently, the catalog went through a writers' workshop process. Five researchers from the pattern community had two weeks before the writers' session to meticulously review the paper and provide detailed comments for each defined smell. The catalog was extensively discussed during three sessions, each lasting two hours. These sessions involved a thorough examination of each smell, including their definition and concrete examples. Furthermore, the results of this study and our previous work indicate that the studied smells are related to bugs. From the commit messages, we also found that some smells were explicitly discussed by developers who contributed to the smelly files. For example, a developer reported issues related to the smell Not Handling Exceptions in *Conscrypt*: `‘‘JNI call made without checking exceptions when required to from CallObjectMethod’’`).

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 sizes. 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. We studied a particular subset of multi-language design smells. Future works should consider analyzing other sets of multi-language design smells.

Threats to reliability validity To mitigate this threat we provide in this paper all the details needed to fully replicate our study. We used open-source projects available in GitHub. We made all the scripts and data available online in our GitHub repository.[12]

7 Conclusion

In this paper, we examine the impact of multi-language design smells on software bug-proneness. We detected 15 types of multi-language smells from eight open-source projects. We performed a survival analysis and compared the time until a bug occurred 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 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 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. 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. 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.

As part of our future work, we plan to (1) investigate whether the time to fix a bug occurring in smelly files is higher than the time needed to fix the bug in non-smelly files, (2) study the co-occurrence of multi-language smells and traditional smells (that can occur in components written in a single language), (3) study other types of FFI and other combinations of programming languages, and (4) extract a taxonomy of multi-language bugs along with their root causes and recommend mitigation strategies.

References

1. Abbes, M., Khomh, F., Gueheneuc, Y.G., Antoniol, G.: 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, pp. 181–190. IEEE (2011)
2. Abidi, M., Grichi, M., Khomh, F.: Behind the scenes: developers' perception of multi-language practices. In: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, pp. 72–81. IBM Corp. (2019)
3. Abidi, M., Grichi, M., Khomh, F., Guéhéneuc, Y.G.: Code smells for multi-language systems. In: Proceedings of the 24th European Conference on Pattern Languages of Programs, p. 12. ACM (2019)
4. Abidi, M., Khomh, F., Guéhéneuc, Y.G.: Anti-patterns for multi-language systems. In: Proceedings of the 24th European Conference on Pattern Languages of Programs, p. 42. ACM (2019)
5. Abidi, M., Rahman, M.S., Openja, M., Khomh, F.: Are multi-language design smells fault-prone? an empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM) **30** (2020)
6. Abidi, M., Rahman, M.S., Openja, M., Khomh, F.: Are multi-language design smells fault-prone? an empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM)(to appear) (2020). URL `https://arxiv.org/abs/2010.14331`
7. Alexander, C., Ishikawa, S., Silverstein, M., i Ramió, J.R., Jacobson, M., Fiksdahl-King, I.: A pattern language. Gustavo Gili (1977)
8. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G.: 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, pp. 304–318 (2008)
9. Asuncion, H.U., Asuncion, A.U., Taylor, R.N.: Software traceability with topic modeling. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, pp. 95–104. IEEE (2010)
10. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. Journal of machine Learning research **3**(Jan), 993–1022 (2003)
11. Borrelli, A., Nardone, V., Di Lucca, G.A., Canfora, G., Di Penta, M.: Detecting video game-specific bad smells in unity projects. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 198–208 (2020)
12. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998)
13. Castelluccio, M., An, L., Khomh, F.: An empirical study of patch uplift in rapid release development pipelines. Empirical Software Engineering **24**(5), 3008–3044 (2019). DOI 10.1007/s10664-018-9665-y. URL `https://doi.org/10.1007/s10664-018-9665-y`
14. Cleves, M., Gould, W., Gould, W.W., Gutierrez, R., Marchenko, Y.: An introduction to survival analysis using Stata. Stata press (2008)
15. Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A.: Detecting code smells using machine learning techniques: are we there yet? In: 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner), pp. 612–621. IEEE (2018)
16. Fisher, L.D., Lin, D.Y.: Time-dependent covariates in the cox proportional-hazards regression model. Annual review of public health **20**(1), 145–157 (1999)
17. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)

19. Geman, S., Geman, D.: Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. IEEE Transactions on pattern analysis and machine intelligence **PAMI-6**(6), 721–741 (1984)
20. Goedicke, M., Neumann, G., Zdun, U.: Object system layer. 5th European Conference on Pattern Languages of Programms (EuroPLoP '2000) (2000)
21. Goedicke, M., Neumann, G., Zdun, U.: Message redirector. 6th European Conference on Pattern Languages of Programms (EuroPLoP '2001) (2001)
22. Goedicke, M., Zdun, U.: Piecemeal legacy migrating with an architectural pattern language: A case study. Journal of Software Maintenance and Evolution: Research and Practice **14**(1), 1–30 (2002)
23. Gurcan, F., Cagiltay, N.E.: Big data software engineering: Analysis of knowledge domains and skill sets using lda-based topic modeling. IEEE Access **7**, 82541–82552 (2019)
24. Habchi, S., Rouvoy, R., Moha, N.: On the survival of android code smells in the wild. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 87–98. IEEE (2019)
25. Habibi, M., Popescu-Belis, A.: Keyword extraction and clustering for document recommendation in conversations. IEEE/ACM Transactions on audio, speech, and language processing **23**(4), 746–759 (2015)
26. Hunt, J.: Java for Practitioners: An Introduction and Reference to Java and Object Orientation, 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
27. Jelodar, H., Wang, Y., Yuan, C., Feng, X., Jiang, X., Li, Y., Zhao, L.: Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey. Multimedia Tools and Applications **78**(11), 15169–15211 (2019)
28. Johannes, D., Khomh, F., Antoniol, G.: A large-scale empirical study of code smells in javascript projects. Software Quality Journal **27**(3), 1271–1314 (2019)
29. Jones, T.C.: Estimating software costs. McGraw-Hill, Inc. (1998)
30. Khomh, F., Di Penta, M., Gueheneuc, Y.G.: An exploratory study of the impact of code smells on software change-proneness. In: Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, pp. 75–84. IEEE (2009)
31. Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering **17**(3), 243–275 (2012)
32. Khomh, F., Vaucher, S., Guéhéneuc, Y.G., Sahraoui, H.: A bayesian approach for the detection of code and design smells. In: Quality Software, 2009. QSIC'09. 9th International Conference on, pp. 305–314. IEEE (2009)
33. Kochhar, P.S., Wijedasa, D., Lo, D.: 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, pp. 563–573. IEEE (2016)
34. Kondoh, G., Onodera, T.: Finding bugs in java native interface programs. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, pp. 109–118. ACM, New York, NY, USA (2008)
35. Kontogiannis, K., Linos, P., Wong, K.: Comprehension and maintenance of large-scale multi-language software applications. In: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, pp. 497–500. IEEE (2006)
36. Koru, A.G., El Emam, K., Zhang, D., Liu, H., Mathew, D.: Theory of relative defect proneness. Empirical Software Engineering **13**(5), 473 (2008)
37. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program comprehension in multi-language systems. In: Reverse Engineering, 1998. Proceedings. Fifth Working Conference on, pp. 135–143. IEEE (1998)
38. Lee, B., Hirzel, M., Grimm, R., McKinley, K.S.: Debug all your code: Portable mixed-environment debugging. SIGPLAN Not. **44**(10), 207–226 (2009)
39. Lenarduzzi, V., Saarimäki, N., Taibi, D.: The technical debt dataset. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 2–11 (2019)
40. Li, S., Tan, G.: Finding bugs in exceptional situations of jni programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, pp. 442–452. ACM, New York, NY, USA (2009)
41. Liang, S.: Java Native Interface: Programmer's Guide and Reference, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

42. Lima, R., Souza, J., Fonseca, B., Teixeira, L., Gheyi, R., Ribeiro, M., Garcia, A., de Mello, R.: Understanding and detecting harmful code. In: Proceedings of the 34th Brazilian Symposium on Software Engineering, pp. 223–232 (2020)
43. Lin, D.: Goodness-of-fit tests and robust statistical inference for the Cox proportional hazards model. University of Michigan (1989)
44. Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., Guéhéneuc, Y.G.: 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, pp. 232–243. ACM (2014)
45. Linos, P.K.: 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, pp. 338–341. IEEE (1995)
46. Linos, P.K., Chen, Z.h., Berrier, S., O'Rourke, B.: A tool for understanding multi-language program dependencies. In: Program Comprehension, 2003. 11th IEEE International Workshop on, pp. 64–72. IEEE (2003)
47. Long, F., Mohindra, D., Seacord, R.C., Sutherland, D.F., Svoboda, D.: Java coding guidelines: 75 recommendations for reliable and secure programs. Addison-Wesley (2013)
48. McCallum, A.: A machine learning for language toolkit; 2002. URL: http://mallet. cs. umass. edu/[accessed 2015-04-05][WebCite Cache ID 6XZgiQKil] (2019)
49. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: icsm, pp. 120–130 (2000)
50. Morales, R., McIntosh, S., Khomh, F.: Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 171–180 (2015). DOI 10.1109/SANER.2015.7081827
51. Muse, B.A., Rahman, M.M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G.: 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, pp. 327–338 (2020)
52. Mushtaq, Z., Rasool, G.: Multilingual source code analysis: State of the art and challenges. In: Open Source Systems & Technologies (ICOSST), 2015 International Conference on, pp. 170–175. IEEE (2015)
53. Mushtaq, Z., Rasool, G.: Multilingual source code analysis: State of the art and challenges. In: 2015 International Conference on Open Source Systems Technologies (ICOSST), pp. 170–175 (2015)
54. Neitsch, A., Wong, K., Godfrey, M.W.: Build system issues in multilanguage software. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 140–149. IEEE (2012)
55. Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N.: 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, pp. 390–400. IEEE Computer Society (2009)
56. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 101–110. IEEE (2014)
57. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A.: Mining version histories for detecting code smells. IEEE Transactions on Software Engineering **41**(5), 462–489 (2014)
58. Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A.: The scent of a smell: An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering **44**(10), 977–1000 (2017)
59. Pfeiffer, R.H., Wąsowski, A.: Texmo: A multi-language development environment. In: Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA'12, pp. 178–193. Springer-Verlag, Berlin, Heidelberg (2012)
60. Politowski, C., Khomh, F., Romano, S., Scanniello, G., Petrillo, F., Guéhéneuc, Y.G., Maiga, A.: A large scale empirical study of the impact of spaghetti code and blob

anti-patterns on program comprehension. Information and Software Technology **122**, 106278 (2020). DOI https://doi.org/10.1016/j.infsof.2020.106278. URL `https://www.sciencedirect.com/science/article/pii/S0950584920300288`
61. Porter, M.F.: Snowball: A language for stemming algorithms (2001)
62. Radu, A., Nadi, S.: A dataset of non-functional bugs. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 399–403. IEEE (2019)
63. Ray, B., Posnett, D., Filkov, V., Devanbu, P.: 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, pp. 155–165. ACM (2014)
64. Romano, D., Raila, P., Pinzger, M., Khomh, F.: Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In: Reverse Engineering (WCRE), 2012 19th Working Conference on, pp. 437–446. IEEE (2012)
65. Rosen, C., Shihab, E.: What are mobile developers asking about? a large scale study using stack overflow. Empirical Software Engineering **21**(3), 1192–1223 (2016)
66. Saboury, A., Musavi, P., Khomh, F., Antoniol, G.: An empirical study of code smells in javascript projects. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pp. 294–305. IEEE (2017)
67. Samoladas, I., Angelis, L., Stamelos, I.: Survival analysis on the duration of open source projects. Information and Software Technology **52**(9), 902–922 (2010)
68. Selim, G.M., Barbour, L., Shang, W., Adams, B., Hassan, A.E., Zou, Y.: Studying the impact of clones on software defects. In: 2010 17th Working Conference on Reverse Engineering, pp. 13–21. IEEE (2010)
69. Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.G.: 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, pp. 393–402. IEEE (2016)
70. Spadini, D., Aniche, M., Bacchelli, A.: 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, pp. 908–911. ACM (2018)
71. Tan, C.M., Wang, Y.F., Lee, C.D.: The use of bigrams to enhance text categorization. Information processing & management **38**(4), 529–546 (2002)
72. Tan, G., Chakradhar, S., Srivaths, R., Wang, R.D.: Safe Java Native Interface. In: In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering, pp. 97–106 (2006)
73. Tan, G., Croft, J.: An empirical security study of the native code in the jdk. In: Proceedings of the 17th Conference on Security Symposium, SS'08, pp. 365–377. USENIX Association, Berkeley, CA, USA (2008)
74. Thomas, S.W.: Mining software repositories using topic models. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1138–1139 (2011)
75. Thongtanunam, P., Hassan, A.E.: Review dynamics and their impact on software quality. IEEE Transactions on Software Engineering (2020)
76. Till, Q.: How to ship product with a quarterly product roadmap and sprint-based execution. In: website (2019). URL `https://www.getshipit.com/blog/how-to-ship-product-with-a-quarterly-product-roadmap/`
77. Tomassetti, F., Torchiano, M.: An empirical assessment of polyglot-ism in github. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, pp. 17:1–17:4. ACM, New York, NY, USA (2014)
78. Treude, C., Wagner, M.: Predicting good configurations for github and stack overflow topic models. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 84–95 (2019)
79. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)
80. Yamashita, A., Moonen, L.: Do code smells reflect important maintainability aspects? In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 306–315. IEEE (2012)

81. Yamashita, A., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251. IEEE (2013)
82. Yin, R.K.: Applications of Case Study Research Second Edition (Applied Social Research Methods Series Volume 34). {Sage Publications, Inc} (2002)