Noname manuscript No.
(will be inserted by the editor)

Multi-language Design Smells: A Backstage Perspective

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

the date of receipt and acceptance should be inserted later

Abstract Multi-language systems became prevalent with technological advances. Developers opt for the combination of programming languages to build a single application. Such combinations of programming languages allow the reuse of existing code and libraries without re-implementing the code from scratch. Software quality is achieved by following good software development practices and avoiding the bad ones. However, most of the practices in the literature apply to mono-language systems only and do not consider the interaction between programming languages. We previously defined a catalog of bad practices *i.e.,* design smells related to multi-language systems. This paper aims to provide empirical evidence on the relevance of multi-language design smells and their perceived impacts on software quality. We analysed eight open source projects to detect occurrences of 15 types of multi-language design smells. We also extracted information about the developers that contributed to those systems. We performed an open and a closed survey targeting developers in general but also developers who contributed to those systems. We surveyed developers about the perceived prevalence of multi-language design smells their severity, and their impact on software quality attributes. We report that most of the studied design smells are perceived as design or

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

implementation problems. Our results suggest that the studied design smells could be introduced mainly during refactoring and maintenance activities, and during regular development tasks. Our results also point that multi-language design smells are perceived as harmful and have negative impacts on software quality. The perceived prevalence of design smells and their impact varies from one specific smell type to the others. We believe that our findings are important for developers and researchers interested in improving the quality of multi-language systems as it can help them prioritize design smells during maintenance activities.

Keywords Survey, Multi-language systems, Design smells, JNI

1 Introduction

Design smells reflect symptoms of poor design and implementation choices that may potentially have negative impacts on software quality. Such poor design choices may lead to a variety of maintenance challenges and issues, including the increase of maintenance activities and the introduction of bugs [60,59,22,50,21,49,52]. While a design smell may not definitively identify an error, its presence suggests a potential trouble spot, a place where there is an increased risk of future bugs or potential failures. In the last decade, empirical studies reported that design smells hinder software comprehensibility and may increase bug and change-proneness [21,50]. However, existing studies on design smells primarily focus on mono-language systems and do not consider the smells related to the interaction between programming languages (*i.e.,* multi-language design smells).

On the other hand, multi-language systems are gaining popularity. Today, a common practice to develop an application is to combine components written in different programming languages and technologies [29,44,26]. Such practice allows to gain the benefits from the strengths offered by each programming language and to reuse existing code. Several studies in the literature have investigated challenges related to multi-language systems [26,30,44, 26,16,17,46]. The majority of these studies report that program comprehension and system's complexity are the main challenges of the quality assurance of multi-language systems [44,26,39]. Software quality is partially achieved by adopting formal guidelines, design patterns, and avoiding design smells [62,23, 24]. However, despite the increasing popularity of multi-language systems, the literature is still lacking an established set of guidelines, *i.e.,* design patterns and design smells to follow or avoid when combining different programming languages in order to maximise their benefits [46,26,4]. The information about multi-language design patterns, design smells, and practices (*e.g.,* [46,17,55]) is scattered in different resources [40]. Developers and researchers may not easily access these resources. This diversity of sources of information makes it difficult for developers to clearly identify which practices to adopt and which ones to avoid (to benefit from using multiple programming languages). This diversity also adds additional challenges to the development and maintenance of multi-language systems. Developers working on any part of the system are required to have experience in multiple programming languages. Moreover, they should consider the compatibility and rules (*i.e.,* semantic, lexical, and syntactical) related to each programming language to correctly handle the inter-language communications.

We previously documented a catalog of 15 multi-language design smells, *i.e.,* recurrent bad coding or design practices when combining programming languages [40,42]. These definitions and characteristics of design smells have been extracted from several sources of information (*e.g.,* developers blogs, literature, bug reports, and source code) and validated through rounds of shepherding process and writers' workshop of a pattern conference (EuroPlop). During this validation process, the design smells were discussed along with their definition and concrete examples with experts on patterns and smells. In a recent study, we performed an empirical investigation of the impact of the design smells on software fault-proneness [6]. From our analysis, we found that design smells are prevalent and that they have a negative impact on the software fault-proneness. However, these assertions have never been verified with professional developers. Therefore, we aim through this paper to complement our previous work [6], and assess the perception of the defined design smells from the developers' perspective. We also aim to investigate developers' perception of the severity of the multi-language design smells and their impacts on software quality attributes. Our five key contributions are: (1) empirical investigation of the developers' perception about multi-language design smells in the context of JNI systems, (2) text-based analysis to identify the reasons behind the introduction of design smells, (3) empirical evaluation of the perceived severity of multi-language design smells, (4) their perceived impact on software quality attributes, and (5) investigation of the possible refactoring solutions related to design smells occurrences.

To achieve these objectives, we started by analysing the source code of open source projects to extract occurrences of multi-language design smells in the context of JNI systems. We analysed 270 snapshots of eight open source projects. We also extracted information about the developers who contributed to the files impacted by the design smells. We then designed two surveys: an open survey targeting professional software developers and a closed survey targeting developers who contributed to files containing the studied multi-language design smells. We use surveys because it is a "system for collecting information from or about people to describe, compare, or explain their knowledge, attitudes, and behavior" [13]. We believe that surveying developers is the best method to retrieve developers' perceptions about multi-language design smells, since they are the ones who interact daily with these systems and who suffer from their challenges. More specifically, we surveyed developers about their perceived prevalence, severity, and impacts of the studied design smells on software quality attributes. We received a total of 132 responses for the open survey and 39 responses for the closed survey.

The analysis of surveys responses shows that (1) overall, developers consider the proposed design smells to be reflective of design and implementation problems. (2) The main reasons for smells introduction, reported by the participants are: refactoring and maintenance, continuous development (*i.e.,* perform regular development tasks), easy way of implementation, lack of knowledge, and specific implementation and design choices. (3) The design smells are perceived in general to negatively impact all the studied quality attributes. (4) The design smells perceived as the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, and *Excessive Inter-language Communication.* (5) In general, developers would consider refactoring the design smells from their systems.

The remainder of this paper is organised as follows. Section 2 discusses the background of multi-language design smells. Section 3 describes the design of our study in general and survey in particular. Section 4 reports results of our survey while Section 5 discusses these results and provides some recommendations. Section 6 summarises threats to the validity of our study. Section 7 presents related work. Section 8 concludes with future works.

2 Background

In this section, we first introduce a brief background on multi-language systems. We then discuss different types of multi-language design smells studied in this paper.

2.1 Multi-language Systems

Modern software systems are moving from the usage of a single programming language towards the combination of programming languages [35,28,26]. These systems are referred to as multi-language systems [11], *i.e.,* software systems that are developed with a combination of components written with at least two programming languages. The languages may have diverse lexical, semantic, and syntactical programming rules. Most of the systems with which we interact daily integrate components written in several programming languages and technologies. Such systems are gaining popularity because of their different inherent benefits [56,48,45,9,14,19,25,53]. For example, it allows developers to reuse existing components and external libraries to reduce the development time and cost [2,55,29]. Developers often leverage the strengths of different programming languages to cope with the pressure and the challenges of building complex systems [56,48,45].

2.2 Java Native Interface

While some applications could be developed completely in Java, there are situations where Java alone cannot offer the desired level of functionality or performance (*e.g.,* speed) and 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 [33,20]. We present in Figure 1

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

(a) JNI Method Declaration

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


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

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

}
```

(b) JNI Implementation Function

Fig. 1: JNI HelloWorld Example

an example of a JNI code extracted from [33]. Figure 1 (a) presents a Java class that contains a native method declaration `Print()` and loads the corresponding native library while Figure 1 (b) presents the C file that contains the implementation of the native function `Print()`. `JNIEXPORT` and `JNICALL` are the macros needed to link the native method declaration in Java with its corresponding implementation in C [33].

2.3 Multi-language Design Smells

Multi-language design smells are defined as poor design and implementation choices when combining different programming languages. They may slow down the development process of multi-language systems or increase the risk of bugs or potential failures in the future [3,4]. A few papers in the literature discussed the design patterns and design smells related to multi-language systems. An extensive catalog for multi-language design smells was published by Abidi *et al.* [3,4]. In the following, we elaborate on each of the design smells studied in this paper; providing an illustrative example. More details about these design smells are available in the reference catalog [3,4]. While some of the studied design smells (*e.g., Unused Parameters*, *Too Much Scattering*, and *Too Much Clustering*) could be applied to a single programming language, in this study we are considering only the situations where the design smells occur in the context of multi-language systems. As described in the previously published catalog, multi-language systems are by nature more difficult to understand and introduce additional challenges compared to mono-language systems. Those challenges are mainly related to the incompatibilities of programming languages and the heterogeneity of components. Having such design smells occurring on those systems are expected to increase the challenges related to the maintenance of these systems and introduce additional complexity. It may be difficult for a maintainer to identify the design smells occurrences, to retrace or fix a bug across different programming languages. The studied design smells involve components and code written in different programming languages and developers may not have a full picture and-or understanding of the whole system.

1. *Not Handling Exceptions* (NHE): The exception handling mechanism is helpful to identify and subsequently to report that an error has occurred. However, such mechanism is programming language dependent. Indeed, the Java exception mechanism is quite self-sufficient. When an exception is thrown in Java code (from try block), the control looks for the appropriate `catch` block that could be used to properly handle the exception. However, unlike Java, when using JNI, the native code (C/C++) does not provide an automatic mechanism for handling exceptions. In the case of JNI applications, 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 [3,55,31,27].[1] Mishandling of JNI exceptions may introduce vulnerabilities and expose the JVM to security breaches [55,31,27]. Listing 1 introduces an example of this type of design smell extracted from *OpenDDS*.[2] In this example, the use of `findClass, GetFieldID` without verifying that the call to these methods was properly executed may lead to vulnerabilities. If the class `clazz` or the field `fid` were not retrieved correctly, this could lead to errors. A suggested solution to this smell is to add a condition statement to verify that these functions were correctly executed or to use the throw exceptions statement using `Throw()` or `ThrowNew()`.

Listing 1: Design Smell - Not Handling Exceptions Across Languages

```
/* C++ */
{
  jclass clazz = findClass(jni, "i2jrt/TAOObject");
  jfieldID fid = jni->GetFieldID(clazz, "_jni_ptr", "J");
  jlong _jni_ptr = jni->GetLongField(jThis, fid);
  CORBA::Object_ptr o = reinterpret_cast<CORBA::Object_ptr>(_jni_ptr);
  CORBA::release(o);
  jni->SetLongField(jThis, fid,
                   reinterpret_cast<jlong>(CORBA::Object::_nil()));
}
```

2. *Assuming Safe Return Value* (ASRV): Since JNI requires the use of some predefined functions to access Java objects from the native C/C++ code, checking return values in the native code is important to ensure the cor-

[1] https://www.ibm.com/developerworks/library/j-jni/index.html

[2] https://github.com/objectcomputing/OpenDDS/blob/945a0df6f4a2e52be9eb766d7b717d146d1649f1/java/idl2jni/runtime/i2jrt_TAOObject.cpp

rectness of the program. 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. Developers should implement checking values before returning variables from the native code to Java code to ensure that the program was correctly executed. Indeed, assuming those values as correct or safe without explicit verification may lead to errors and security issues [31,3]. Listing 2 illustrates an example of this type of design smell extracted from Android Platform.[3] 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. A possible solution would be to implement checks in the native code that handle situations in which problems may occur with the return values.

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

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

3. *Not Securing Libraries* (NSL): Improper use of JNI can render Java applications vulnerable to security flaws in the native code. When using JNI, a popular way to load the native library is the use of the method *loadLibrary* without using a secured block. In such circumstances, the native library is loaded without performing security checks. Therefore, malicious code can take advantage of such security flaws to call native methods from the library. This design smell may negatively impact the security and reliability of the system [37,3]. Listing 3, presents an example of possible refactoring for this type of design smell where the native library is loaded within a secure block extracted from the JDK.[4] It is important to use a secure block to ensure that the libraries cannot be loaded without permission. In the context of JNI systems, it is recommended to always load libraries in static blocks, wrapped in a call to *AccessController.doPrivileged* or use the *securityManager* to ensure that the library cannot be loaded without permission [3].
4. *Hard Coding Libraries* (HCL): When the same code is required to run on various platforms, we need to customize the loading of the native library depending on the operating system. This design smell occurs when the native libraries are loaded without considering operating system specific conditions and requirements (using predefined methods, for example for Java: System.getProperty("os.name")). In such situations, the loading of the library is hard-coded by the developers according to the OS.

[3] `https://android.googlesource.com/platform/frameworks/base/+/ba34751/core/jni/android_opengl_GLES10Ext.cpp`
[4] `https://github.com/oracle/graal/issues/1388`

Listing 3: Securing Library Loading

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

Therefore, it could be difficult for an external developer to know which library is actually loaded, which may cause confusion. We present in Listing 4 an example of the occurrence of this design smell extracted from *java-smt*. For the example presented in Listing 4, there were some issues reported that discuss the possible confusion that this type of design smell could introduce[5]:"`Z3's Native class attempts to load z3java and on failure loads libz3java, we try to guess the OS instead of using a fallback.`", "`However, Z3 uses the Linux naming convention even for its Windows binaries, and thus the loadLibrary("z3") won't find anything on Windows. So we need either the if block, or a fall-back like the Native class does`".

Listing 4: Design Smells - Hard Coding Libraries

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

5. *Not Using Relative Path* (NRP): This design smell is defined by loading the native library using an absolute path of the library instead of the corresponding relative path. Using relative paths, native libraries can be loaded anywhere. However, if the native library is no longer available or relocated, referring to its absolute path can introduce bugs. This may likewise affect the reusability of the code and impact the maintenance activities because the native library can become inaccessible due to an incorrect path.
6. *Too Much Clustering* (TMC): Declaring an excessive number of native methods in the same class is likely to reduce the readability and consequently may impact the maintainability of the code. Such bad coding practice will increase the lines of code in the class and hence will make the code harder to maintain. Several studies have discussed the optimal number of methods to include within the same class, such as the rule of 30 proposed by Martin Lippert [36], or the *7 plus/minus 2 rule* stating that the human mind can accommodate and understand from five to nine objects. The majority of the discussed relevant measures are the single principle

[5] `https://github.com/sosy-lab/java-smt/issues/87`

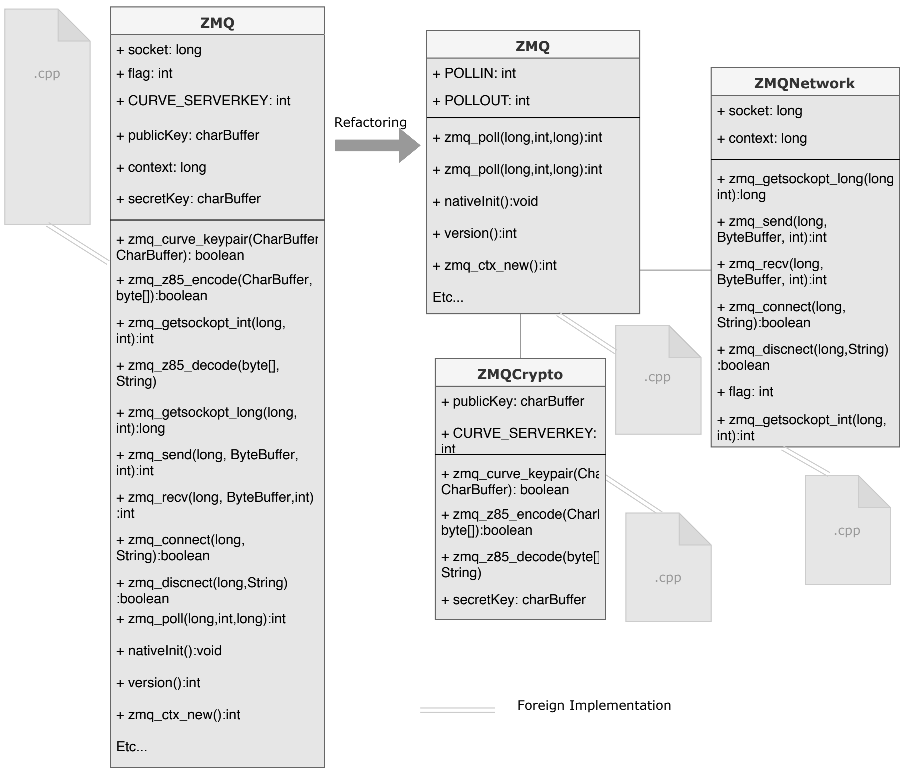

.cpp
ZMQ
+ socket: long
+ flag: int
+ CURVE_SERVERKEY: int
+ publicKey: charBuffer
+ context: long
+ secretKey: charBuffer
+ zmq_curve_keypair(CharBuffer, CharBuffer): boolean
+ zmq_z85_encode(CharBuffer, byte[]):boolean
+ zmq_getsockopt_int(long, int):int
+ zmq_z85_decode(byte[], String)
+ zmq_getsockopt_long(long, int):long
+ zmq_send(long, ByteBuffer, int):int
+ zmq_recv(long, ByteBuffer,int) :int
+ zmq_connect(long, String):boolean
+ zmq_discnect(long,String) :boolean
+ zmq_poll(long,int,long):int
+ nativeInit():void
+ version():int
+ zmq_ctx_new():int
Etc...
Refactoring
ZMQ
+ POLLIN: int
+ POLLOUT: int
+ zmq_poll(long,int,long):int
+ zmq_poll(long,int,long):int
+ nativeInit():void
+ version():int
+ zmq_ctx_new():int
Etc...
ZMQNetwork
+ socket: long
+ context: long
+ zmq_getsockopt_long(long int):long
+ zmq_send(long, ByteBuffer, int):int
+ zmq_recv(long, ByteBuffer, int):int
+ zmq_connect(long, String):boolean
+ zmq_discnect(long,String) :boolean
+ flag: int
+ zmq_getsockopt_int(long, int):int
ZMQCrypto
+ publicKey: charBuffer
+ CURVE_SERVERKEY: int
+ zmq_curve_keypair(Cha CharBuffer): boolean
+ zmq_z85_encode(Charl byte[]):boolean
+ zmq_z85_decode(byte[] String)
+ secretKey: charBuffer
.cpp
.cpp
.cpp
Foreign Implementation

Fig. 2: Illustration of Design Smell - Too Much Clusterings

responsibility, high cohesion, low coupling, and the separation of concerns. In this context, a bad practice would be to concentrate multi-language native methods in a few classes, regardless of their roles and responsibilities, leading to huge classes with many methods and thus low cohesion. Figure 2 provides an example of this design smell extracted from ZMQJNI.[6] In this example, native methods are declared within the same class. However, some of those native methods are related to cryptographic operations while others are related to network communication. This merging of concerns resulted in a blob multi-language class with 29 native declaration methods and 78 attributes. Another example is the class `NativeCrypto` in *Conscrypt* in which 286 native methods are declared in the same class regardless their concerns.[7]

7. *Too Much Scattering* (TMS): Managers and developers often need to settle on a balance between isolating native code within a single or a few classes or splitting it between many classes [3]. Getting to this compromise is estimated to improve the readability and maintainability of the project

[6] `https://github.com/zeromq/zmq-jni/commit/6b0d25dd45de42680c965ec8c27f7596661bb6fa`

[7] `https://github.com/google/conscrypt/blob/23c2ce258afd5a43258b685df4279eaa2ee0e15c/common/src/main/java/org/conscrypt/NativeCrypto.java`

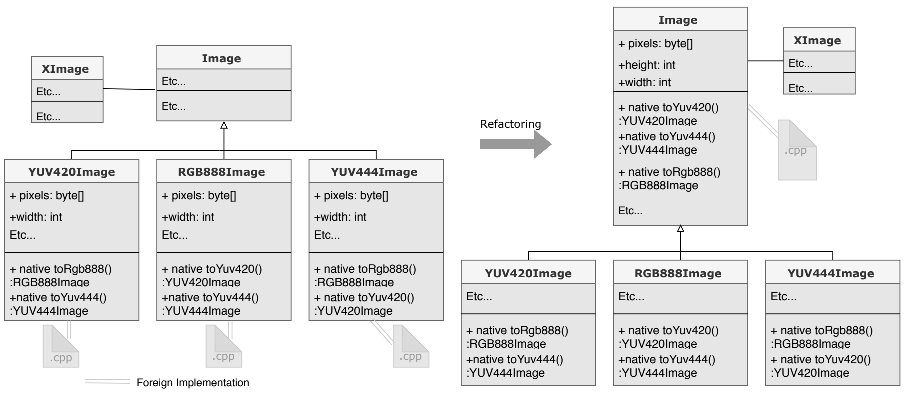

XImage
Etc...
Etc...
Image
Etc...
Etc...
YUV420Image
+ pixels: byte[]
+width: int
Etc...
+ native toRgb888() :RGB888Image
+native toYuv444() :YUV444Image
RGB888Image
+ pixels: byte[]
+width: int
Etc...
+ native toYuv420() :YUV420Image
+native toYuv444() :YUV444Image
YUV444Image
+ pixels: byte[]
+width: int
Etc...
+ native toRgb888() :RGB888Image
+ native toYuv420() :YUV420Image
.cpp
Foreign Implementation
Refactoring
Image
+ pixels: byte[]
+height: int
+width: int
+ native toYuv420() :YUV420Image
+native toYuv444() :YUV444Image
+ native toRgb888() :RGB888Image
Etc...
XImage
Etc...
Etc...
.cpp
YUV420Image
Etc...
+ native toRgb888() :RGB888Image
+native toYuv444() :YUV444Image
RGB888Image
Etc...
+ native toYuv420() :YUV420Image
+native toYuv444() :YUV444Image
YUV444Image
Etc...
+ native toRgb888() :RGB888Image
+ native toYuv420() :YUV420Image

Fig. 3: Illustration of Design Smell - Too Much Scattering

[3]. This design smell is the opposite to smell type *Too Much Clustering*. While the design smell *Too Much Clustering* could be introduced when multi-language code is concentrated in a single or a few classes regardless of their concerns, the design smell *Too Much Scattering* , on the other hand, is introduced when the native code is dispersed between many classes without satisfying both the coupling and the cohesion. Figure 3 presents an example of design smell of type *Too Much Scattering* extracted from a previous work [4]. This example presents classes that contain only two native method declarations. The methods and classes participating in the multi-language interaction are spread through the code in a way that determining which classes are participating in the multi-language interaction requires some effort. Such code will be more difficult to maintain and refactor [4]. A suggested solution is to regroup the duplicated native methods in the same class to reduce the number of classes participating in the multi-language code.

8. *Excessive Inter-language Communication* (EXC): A wrong partitioning in components written in different programming languages results in many calls in one way or the other. This may increase the complexity and may also affect the performance. The design smell *Excessive Inter-language Communication* may indicate a bad separation of concerns between different layers or components that are implemented in different programming languages. For instance, the same object could be modified by multiple components written in different programming languages. An excessive call to the native code from the host code, could be introduced by calling several native methods within the same class or within a loop with a large number of iterations. In such situations, an object is passed several times to and from the native code which may result in excessive inter-language communications.
9. *Local References Abuse* (LRA): A local reference is created for any object that is returned by the native code. For each method, JNI specification permits up to 16 local references. Therefore, developers should be care-

ful when dealing with Java objects and pay attention to the number of references that are created. It is also recommended to always delete unnecessary local references using `JNIDeleteLocalRef()`. Listing 5 presents an example of this design smell, where local references are created using the method *GetObjectArrayElement()* but not deleted accordingly by calling any of the memory release methods JNIDeleteLocalRef() or RelaseObjectArrayElement().

Listing 5: Design Smell - Local References Abuse

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

10. *Memory Management Mismatch* (MM): The data types are different between Java and native code.[8] JNI treats Java objects and classes as reference types. The JVM provides a bunch of predefined methods that can be used to access Java objects, fields, and methods from the native code. Those predefined methods used allow native code to either return a pointer to the actual element at run-time or to allocate memory and create a copy of that element. Therefore, because of the differences of types between Java and native code, the memory will be allocated to perform corresponding type mapping between Java and C/C++. The allocated memory should then be released after usage, if not, memory leaks will occur. Listing 6 illustrates an example of the occurrence of this design smell extracted from JNA[9] in which the memory was allocated to access a Java Byte Array but was not released using `ReleaseByteArrayElements()`.

Listing 6: Refactoring - Memory Management Mismatch

```
/* C++ */
                 jlong* data = (*env)->GetLongArrayElements(env, handles, NULL);
  int count = (*env)->GetArrayLength(env, handles);
```

11. *Not Caching Objects* (NCO): To access the fields of Java objects and invoke their methods from the native code through JNI, the native code must use predefined functions *i.e., FindClass(), GetFieldId(), GetMethodId(), and GetStaticMethodId()*. For a given class, IDs returned by calling *GetFieldId(), GetMethodId(), and GetStaticMethodId()* remain unchanged during the lifetime of the JVM cycle. The invocation of these methods is very expensive as it may require critical work in the JVM. In this case, it is recommended for a given class to look up the IDs only during the first usage and keep the reference for any future use. In a similar setting, looking up class objects can be expensive. A recommended practice is to cache commonly used classes, field IDs, and method IDs globally, so they

[8] `https://www.developer.com/java/data/jni-data-type-mapping-to-cc.html`
[9] `https://github.com/java-native-access/jna/commit/a77f47fb297dd33a8cec4c88ae4777186882f472`

can be reused when needed. Listing 7 gives an illustration of the design smell *Not Caching Objects* that does not use cached field IDs extracted from Developers' documentation.[1]

Listing 7: Design Smell - Not Caching Objects' Elements

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

12. *Excessive Objects* (EO): Accessing the elements of a field by passing the entire object is a common practice in object-oriented programming. Notwithstanding, in the context of JNI, because the Object type does not exist in C language programs, passing too many objects could prompt additional overhead to properly perform type conversion. Indeed, this design smell occurs when developers pass the entire object as a parameter although only some of its fields were required, and it would have been better for the system performance to pass only those fields (except for the purpose of passing the object to native code to set its elements using *SetxField* method). In the context of object-oriented programming, a good solution is to pass objects for better encapsulation. However, in the context of JNI, native code must reach back to the JVM through multiple calls to obtain the value of each field. This may add additional overhead and likewise may increase the lines of code which may affect the readability of the code [3]. Listing 8 illustrates an example of occurrence of the design smell passing excessive objects extracted from Developers' documentation.[1] A possible solution to this smell would be to pass the class's fields in the method signature as parameters as described in our published catalog [3].

Listing 8: Design Smell - Passing Excessive Objects

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

13. *Unused Method Implementation* (UMI): This smell appears when a method is declared in the host language (Java in our case). This method is implemented in the native code (C or C++). This method may never be called from the host language. This could result from the relocation or refactoring in which developers selected keeping those methods to avoid breaking any

connected code or features. In the context of multi-language code, developers may not easily locate such methods. Such systems usually employ different teams working with different parts of the code. Therefore, it is not an easy task to know which methods are used or not in the foreign code.

14. *Unused Method Declaration* (UMD): This design smell and the previous one are very similar. Nonetheless, they contrast in the implementation part, while for the smell *Unused Method Implementation*, the method is implemented yet never called, in the case of the occurrence of the smell *Unused Method Declaration*, the unused method is declared, however, not implemented and never called. Such methods could remain in the project for an extensive stretch of time without being eliminated on the grounds that having them do not present any bug when executing the program. However, they may contrarily affect the maintenance activities and effort required to maintain those classes.
15. *Unused Parameters* (UP): Extensive list of parameters make methods difficult to understand [15]. It could likewise be an indication that the method is doing too much or that a portion of the parameters is not, at this point utilized. With regards to multi-language programming, a few parameters might be part of the method signature even if they are no longer used by components developed in other programming languages. Since multi-language systems involve developers from various teams, those developers often avoid removing such parameters since they may not be certain if those parameters are used by different components. It could be a challenging task for a specific developer to know what part of the code is used by the foreign code. Listing 9 presents a representation of this design smell extracted from Conscrypt[10]where not all the parameters that are part of the native method signature are in fact used in the implemented function.

Listing 9: Design Smell - Unnecessary Parameters

```
/* C++ */
static jint NativeCrypto_get_X509_ex_flags(JNIEnv* env, jclass, jlong x509Ref,
                                           CONSCRYPT_UNUSED jobject holder) {
    CHECK_ERROR_QUEUE_ON_RETURN;
    X509* x509 = reinterpret_cast<X509*>(static_cast<uintptr_t>(x509Ref));
    JNI_TRACE("get_X509_ex_flags(%p)", x509);

    if (x509 == nullptr) {
        conscrypt::jniutil::throwNullPointerException(env, "x509 == null");
        JNI_TRACE("get_X509_ex_flags(%p) => x509 == null", x509);
        return 0;
    }
}}
```

[10] https://github.com/google/conscrypt/blob/master/common/src/jni/main/cpp/conscrypt/native_crypto.cc

3 Study Design

In this section, we present the methodology of our study.

3.1 Setting Objectives and Research Questions

We started by setting the objective of our study. Our objectives are to assess the perceived prevalence of multi-language design smells and their severity. We also aim to investigate how the impact of those design smells is perceived by developers. We chose to carry out an empirical study using a survey because development and maintenance are manual activities performed by the developers. Developers' perception of the prevalence and impact of multi-language design smells is important since they are the ones who maintain the multi-language systems and who suffer from the associated challenges. In this study, the quality focus is source code comprehension and maintainability, which can be negatively impacted by the occurrences of design smells. The context of the study consists of (i) objects, *i.e.,* design smells detected in open source projects; and (ii) subjects (developers), *i.e.,* professional developers sharing their perception of multi-language design smells. Our target subjects in this study are software developers in general but also the original developers who contributed to the analyzed systems. We defined our research questions as follows:

RQ1: **Are multi-language design smells commonly faced by developers?** We previously documented a catalog of multi-language design smells [40, 42]. We aim through this research question to assess how developers perceive the prevalence of this catalog. Our goal is to capture the opinions of software developers in general, but also the opinion of the specific group of developers who contributed to the code impacted by those smells. We are also interested to investigate if some specific types of smells are perceived to be more prevalent than others. We defined the following null hypothesis H_1*: The design smells studied in this paper are not commonly faced by developers.*

RQ2: **What are the reasons behind introducing multi-language design smells?** In order to better support developers in improving the quality of multi-language systems, it is important to understand the circumstances under which particular design smells occur. Occurrences of design smells could have been intentionally introduced as a result of poor knowledge about multi-language practices. Thus, we aim to study the rationale behind introducing such smells. Our goal here is to study the reasons that could lead to the introduction of those smells and under what circumstances developers are more prone to introducing such smells. In particular, we test the hypothesis H_2*: There are no specific reasons behind the introduction of multi-language design smells.*

RQ3: **What is the perceived impact of multi-language design smells?** We aim to study the perceived impact of multi-language design smells, on a selected set of software quality attributes. These design smells were defined and cataloged based on several sources of information (*e.g.,* developers blogs, literature, bug reports, and source code) [4,3]. In our previous study [6], we have observed that these design smells increase the risk of faults in software systems. In this research question, we aim to understand the perception of developers regarding the severity of the proposed smells. This is important, since the urgency with which they refactor the smells is likely to be affected by their perception of the importance of the issues posed by the design smells. We defined the following null hypothesis H_3*: Software quality attributes are not impacted by the studied smells.*

RQ4: **What are the design smells that developers perceive as the most harmful?** During maintenance activities, developers are interested to identify parts of the code that should be tested in priority or refactored. Hence, we aim to identify design smells that are perceived by developers as the most critical, *i.e.,* making the project more prone to faults or increase maintenance costs. We defined the following hypothesis H_4*: Developers perceive the design smells to have equal impacts on multi-language systems.*

RQ5: **Do developers plan to refactor those design smells?** Design smells are generally associated with a specific set of refactoring strategies depending on the type of smell. Depending on the development and maintenance costs, developers may decide whether or not to remove some specific smells. Therefore, we aim to investigate to what extent developers would apply such refactoring. We defined the following hypothesis H_5*: There is no specific strategy that developers would adopt to refactor multi-language design smells*

3.2 Study Context

To achieve our objectives and to answer our research questions, we started by analyzing the source code of selected open source projects.

Material and Objects The objects considered in this study are the occurrences of 15 types of multi-language design smells detected in eight open source projects. We analyzed a total of 270 snapshots. We selected these projects because they are well-maintained and active projects on GitHub. Another criterion for the selection was 'diversity', *i.e.,* those systems are from diverse application domains, of different sizes, with varying distributions of Java and C/C++ code. Table 1 summarizes the characteristics of the selected systems.

We reused the `MLSInspect` approach proposed in our previous study to detect the occurrences of multi-language design smells [6]. This approach is able to detect occurrences of those smells only in the context of JNI systems (Java and C/C++). We previously evaluated this detection approach and measured the recall and precision [6]. To ensure the reliability of our surveys and to mitigate any possible threats related to the recall and/or precision of the detection approach, we manually validated all the occurrences of design smells that were

Table 1: Overview of the Studied Systems

Projects	Domain	#Snap	LOC	Java	C/C++
Rocksdb	Facebook Database	36	487,853	11%	83.1%
Frostwire	File and Media Sharing	18	403,106	71.4%	19%
Realm	Mobile Database	29	17,1705	82%	8.1%
Conscrypt	Cryptography (Google)	32	91,765	85.3%	14%
Pljava	Database	35	71,910	67%	29.7%
Javacpp	Compiler	30	28,713	98%	0.6%
JNA	Native Shared Library	32	590,208	70.2%	15.4%
OpenDDS	Adaptive Communication	58	2,803,49	5%	16%

used in our survey questions to developers. This manual validation allowed us to remove any false positives. Note that, we did not retrieve all instances of the studied smells in each system. The manual validation was performed to select representative examples for the surveys that are not ambiguous and easy to understand for JNI developers. The validation process was performed by the authors that defined the design smells. We confirmed that the examples provided in our study follow the definition and rules of the smells types used when documenting the design smells [3,4,6].

Participants The survey described in this study is a combination of both an open and a closed survey. We present in the following our methodology to gather responses from the participants, *i.e.,* developers for each of those surveys. For the open survey, we collected developers' backgrounds and demographic information to deal with the representativeness of the results. We followed guidelines of sampling methods to build samples that seek to meet the goals of this study to ensure better representativeness [18,8].

- **Open Survey:** We used convenience sampling and randomly contacted developers that satisfy the criteria of the study. We used LinkedIn[11] as a research tool to reach potential developers. Similar to previous work, we combine different sampling strategies [8]. By convenience sampling we consider available developers willing to participate in the survey. Since in this study we present code snippets only for JNI systems, we target developers having experience with both Java and C/C++ programming languages.
- **Closed Survey:** We mined open source projects from GitHub repository and detected occurrences of multi-language design smells as described earlier using `MLSInspect` [6]. We also collected information about the developers that contributed to the smelly files based on the commit logs. By 'contributor' we refer to any developer who did at least a commit on those files. Note that in the closed survey, a developer is only asked about smells contained in the files that he contributed to.

We decided to run both an open and a closed survey to capture not only the perception of developers who contributed to the code of smelly files, but

[11] `https://www.linkedin.com/`

also the perception of other developers who are less familiar with the smelly code files.

Studies in the literature discussed the ethics of sampling strategies [8]. This study was subject to ethical approval from the Research Ethics Board of Polytechnique Montreal which regulates the ethical and scientific criteria designed to protect human participants[12].

3.3 Study Procedure

The experimental protocol consists of surveys that developers had to answer through the *CheckMarket* website[13]. CheckMarket allows conducting fully-anonymous surveys while keeping track of developers by generating tokens in the form of Ids for each participant. We combine in this study both open and closed surveys[14]. The open survey is targeting software developers in general while the closed survey is targeting developers extracted from commit logs and identified as contributors in the smelly files. We prepared the surveys based on our literature review of the state-of-the-art on multi-language systems and design smells. This review helped us to identify the dimensions and scope of the questionnaire, the individual questions, and the possible answers for each question [38, 27, 31, 47].

Both surveys start with a preamble that includes the information about the principal investigator, the research team, the ethical rights and responsibilities of the investigators and developers (*e.g.,* policy of the study with respect to anonymity). The preamble also describes the objective of the study so that developers understand our motivations for this survey and have full information to decide whether or not to answer the surveys (or parts thereof). In the surveys, we also provide definitions of the concepts and the design smells used in this study.

The open survey consists of three parts, while the closed survey consists of two parts. As shown in Table 2, the open survey contains in the first part four closed questions to collect background information about the developers and their profiles. The survey asks about their work position, their number of years of working experience, the domain of activities of their organisations, and their levels of skills in selected programming languages. These languages were reported to be in the top ten list of languages used world-wide[15]. Note that we did not include the background information in the closed survey since we are not randomly contacting developers but our target population for the closed survey is the original developers who contributed in the object systems. The second part of the open survey (which corresponds to the first part of the closed survey) contains general questions about multi-language design smells. It asks

[12] `https://share.polymtl.ca/alfresco/service/api/node/content/workspace/SpacesStore/b7fbaa9e-8055-41cc-b016-dac345f6cb97?a=false&guest=true`

[13] `https://www.checkmarket.com/`

[14] `https://s-ca.chkmkt.com/?e=189517&h=B445656A9769E90&l=en`

[15] `https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`

Table 2: Survey Questions

N.	Survey Question
Background (Only for the open survey)	
Q1	What is your role within your organization?
Q2	How many years of experience do you have in software engineering?
Q3	What is the domain of activity of your organization?
Q4	What is your level of skill in the following languages?
Section I - Multi-language Design Smells	
Q1	How do you evaluate the impact of those design smells on the following software quality attributes?
Q2	Please rank the following design smells from the harmful to the less harmful
Section II - Specific Tasks for Multi-language Design Smell	
Q1	In your opinion, does the following code fragment(s) contain any occurrence of design smell (implementation and-or design problem)?
Q2	If you answered yes to the previous question, please provide an explanation or specify the design smell(s) involved?
Q3	(In your opinion,) What is the motivation behind using this specific way of implementation?
Q4	Would you apply the following refactoring or would you prefer to keep the initial implementation? please explain.

about the perceived impact on a set of selected software quality attributes, and the perceived severity of each smell type. For the quality attributes, we selected the following four attributes discussed by Gamma et *et al.* in their seminal book about design patterns, *i.e.,* expandability, modularity, reusability, and understandability, and three other attributes, *i.e.,* simplicity, learnability, and performance. We selected these quality attributes because of their relevance for design (anti) patterns and smells [47,23].

As shown in Table 2, we kindly asked the developers in the third part of the open survey (second part of the closed survey) to perform a subset of tasks (Section II of Table 2). We provided in those tasks, the source code snippets and asked the developers about whether or not the presented code snippets contain implementation or design problems, *i.e.,* the design smells. We also asked the developers about the motivation behind that specific implementation and whether developers would consider or not to refactor such code. We proposed a refactored solution and asked the developers whether they would consider or not to apply that refactoring. We also injected code snippets that do not contain any of the design smells to limit the bias of this study, *i.e.,* to isolate situations in which developers may provide arbitrary answers or always indicate the existence of design smells. Following previous work [47], for each type of smell, we randomly selected a representative set of instances to perform the survey. Depending on the type of smells, an instance could be a method, a class, files, or a combination of source code files. We ask about a single smell at the time and do not present code affected by more than one design smell, since we want to evaluate each smell separately. Table 2 reports the survey questions.

Table 3 provides an overview of the smell types and tasks associated with each survey type. The tasks are independent and were designed in a way that allowed us to use survey results even if a participant does not complete the survey until the end; *i.e.,* we can reuse the completed parts of the survey responses. Also note that for the tasks asking about the identification of the smell, all the questions are conditional, *i.e.,* if a participant reports that the

Table 3: Overall Developers' Results For the Smells Identification (Open and Closed Surveys)

Smell	Open	Con	Fro	JNA	Jav	Ope	Plj	Rea	Roc
NHE	57	5	2	0	0	4	3	5	7
UP	63	5	2	1	0	4	3	5	7
NSL	58	5	2	1	4	4	0	5	7
ASRV	71	5	2	0	0	4	0	5	7
LRA	59	4	0	1	0	4	0	0	7
NRP	71	0	2	1	0	3	0	0	3
MM	57	4	0	1	0	0	3	5	5
HCL	72	0	0	0	0	0	0	0	0
EO	57	0	0	0	0	0	0	0	0
NCO	69	0	0	0	0	0	0	0	0
TMC	55	4	2	0	4	3	3	3	4
TMS	71	0	0	0	4	4	3	3	4
EXC	67	3	2	1	3	0	2	0	3
UMD	70	3	2	0	3	4	2	3	4
UMI	54	0	1	0	0	3	2	3	0

Open: OpenSurveys, **Con**: Conscrypt-ClosedSurvey,**Fro**: Frostwire-ClosedSurvey
Jav: JavaCpp-ClosedSurvey, **Ope**: OpenDDS-ClosedSurvey
Plj: PlJava-ClosedSurvey, **Rea**: Realm-ClosedSurvey, **Roc**: Rocksdb-ClosedSurvey

proposed code do not contain any problem or design issue, he will be directly moved to the next task, if not he will be asked to answer the questions related to that task. For the closed survey, as mentioned earlier, a developer was only asked about smells contained in the files in which he contributed code. Not all the developers performed all the survey tasks. For all these reasons, the total number of answers considered in this study for the first part of the survey (Section I of Table 2) is higher than the total number of answers related to each task (Section II of Table 2) as shown in Table 3.

We contacted a total of 500 developers through LinkedIn, we received 132 responses for the open survey. For the closed survey, we contacted a total of 263 developers and received a total of 39 answers. Therefore, for the first part of the survey (Section I), we count a total of 171 answers, while for the other sections, the number varies from one specific type of design smell to the other because, not all the developers responded to all the survey tasks as shown in Table 3.

This study protocol was submitted to Mining Software Repositories (MSR) conference 2020 [5]. The protocol went through a review process and was evaluated by the committee prior to the execution of the survey[16].

3.4 Data Analysis

We present in the following our analysis method to answer the research questions:

For **RQ1**, following previous work [47], we computed for each design smell type, the percentage of answers in which the developers were able to:

[16] `https://osf.io/6yqv5/?view_only=4cca6dc961b44303833917c236e2d667`

- (1) Identify a design or implementation problem, *i.e.,* design smells when we asked them to evaluate the code snippet containing a design smell. By identification, we consider the situations in which the developers selected 'yes' as a response to the question: *In your opinion, does the following code contains any occurrence of design smell (implementation and-or design problem)?*;
- (2) Identify the specific smell that was introduced in the code snippet. By identification, we consider a situation in which the developers provide a correct answer to the question: *If you answered yes to the previous question, please provide an explanation or specify the design smell(s) involved?*

The percentage of correct identifications was computed out of the total number of answers to capture the perceived prevalence of each type of smell from the catalog defined in our previous study [4,3].

For **RQ2**, we used a manual approach of text analysis to extract topics from the answers provided by the respondents to the open questions. This research question consider practitioners answer to the question: *What is your motivation behind using this specific way of implementation?* We relied on the methodology performed by Yamashita and Moonen [60] and used coding techniques to analyse the answer to the open question. We follow a double verification process that was performed independently by two authors to ensure that the answers are in the right categories and that there are no losses of information when combined into categories.

For **RQ3**, we analyzed answers to the closed question: *How do you evaluate the impact of those design smells on the following software quality attributes?* We computed for each type of smell and each quality attribute the percentage of answers in which the developers reported an impact of the smell on that quality attribute. Since the question used the following scale: Positive, Neutral, Negative, and Not applicable (N/A). We have for each smell, (i) the list of quality attributes that are perceived to be negatively impacted by each smell, (ii) the list of quality attributes that are perceived to be not impacted (neutral impact), and (iii) the list of quality attributes, if any, that could be positively impacted by those design smells according to the developers.

For **RQ4**, we analyzed developers' answers to the question: *Please rank the following design smells from the harmful to the less harmful.* Similar to previous work [60], we rely on Borda count to answer this research question. Borda count is a rank-order aggregation technique [12]. If there are n candidates to rank (*i.e.,* design smells in our situation), the first ranked candidates receive n points, the second-ranked receive n-1 points, etc. We obtain a list of all the design smells ranked from the most harmful ones to the less harmful.

For **RQ5**, we compute for each type of design smell presented to the developers (along side a refactoring solution), the percentage of answers in which the developers answered 'yes' to the question *Would you consider using this refactored solution or would you prefer to keep the initial implementation? please explain.*. We proposed three options to the developers *(i) yes, refactor with the proposed solution, (ii) refactor with an alternative solution, and (iii)* *no refactoring.* We also aggregate the answers to provide a general overview of developers' opinions on the application of the proposed refactored solutions.

Fisher's Exact Test: To have further insights into the results, we apply Fisher's exact test [51] to statistically support the results of **RQ1** and **RQ5**. Fisher's exact test is a statistical test designed with the aim to assess if there are non-random associations between two categorical variables. Fisher's exact test checks whether a proportion varies between two different samples. It involves testing the independence of rows and columns in a 2 × 2 contingency table based on the exact sampling distribution of the observed frequencies. In our analysis, we also compute odds ratio (OR) [51] to complement the results of the Fisher's exact test. The OR quantifies the strength of the association between two events of interest. It indicates the likelihood of a particular outcome (*e.g.,* an event occurrence). OR is calculated (as in Equation (1)) as the ratio of the odds p of an event occurring in a sample, *e.g.,* the odds that code snippet with some specific design smells experience the event of interest *e.g.,* correctly identified, considered for refactoring (defined as an experimental group), to the odds q of the same event occurring in another sample, *e.g.,* the odds that code snippet with other types of design smells experience the event of interest, *e.g.,* correctly identified, considered for refactoring (defined as a control group):

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

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

4 Study Results

We now present the results of our survey, answering our research questions.

Demographic information

As mentioned in Section 3, we collected the demographic information only for the open surveys. From the open surveys we received a total of 132 developers' responses. Developers of the open surveys originated from different backgrounds. From our results, we found that 40 (30.3%) are software engineers, 28 (21.2%) are developers, 16 (12.1%) are team leaders, 14 (10.6%) are project managers, 11 (8.33%) are architects, 8 (2.26%) are testers, five (3.79%) are QA-managers, four (2.26%) are self-employed, and 6 (13.53%) selected the option "Other". From our results, we noticed that the developers that answered the survey have mainly five to ten years of working experience (57 out of 132 developers (43.2%)). 43 (32.6%) developers have more than ten years of experience. 30 (22.7%) have from one to five years of working experience, while only two (1.52%) have less than one year of experience. This diversity in the background gives us confidence that we reached a diverse group of developers. From the survey responses, we observe that developers are working in different domains of activities. 38 (28.8%) developers are in research and development, 25 (18.9%) are in analytics (business, IT services, big data), 18 (13.6%) are in networking, 18 (13.6%) are in games, 17 (12.9%) are in robotics and embedded systems, 7 (5.3%) are in banking and insurance. Nine (6.82%) of the developers selected the option "Other". We asked the developers about their levels of skills if they are Novice, have Little Knowledge, Practical, Comfortable, or Expert in C, C++, C#, Java, and Python. The majority reported being expert or comfortable with Java, C, and C++, and comfortable with C#. For Python, most of the developers reported being expert or comfortable. Figure 4 summarizes developers' skills and experience with the selected programming languages.

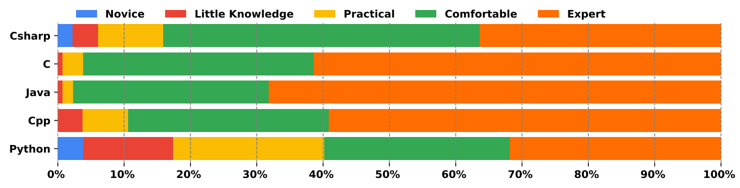


Fig. 4: Developers' Experience With Programming Languages

4.1 RQ1: Are multi-language design smells commonly faced by developers?

Approach Since our goal for this research question is to assess the prevalence of multi-language design smells. For that, we proposed for each smell type an example of code snippet and asked the developers whether they perceive any design or implementation problem. We also asked them to specify the design smell present in that code snippet.

Findings Figure 5a and Figure 5b provide the percentages of developers (for the open and closed surveys, respectively) who (1) reported that there was a design or implementation problem when we asked them to evaluate the code snippet containing a design smell, and (2) correctly identified the smell that was contained in the code snippet, for all the types of smells studied. We report in Table 4 the total number of responses where the smell was correctly identified by the developers for both open and closed surveys (column #Correct), and also the total number of smells that were not correctly identified (column #Not Correct). To give a comparative insight on the proportion of developers who correctly and incorrectly identified the smells, we calculate their corresponding percentages as shown in Table 4 (column %Correct and column

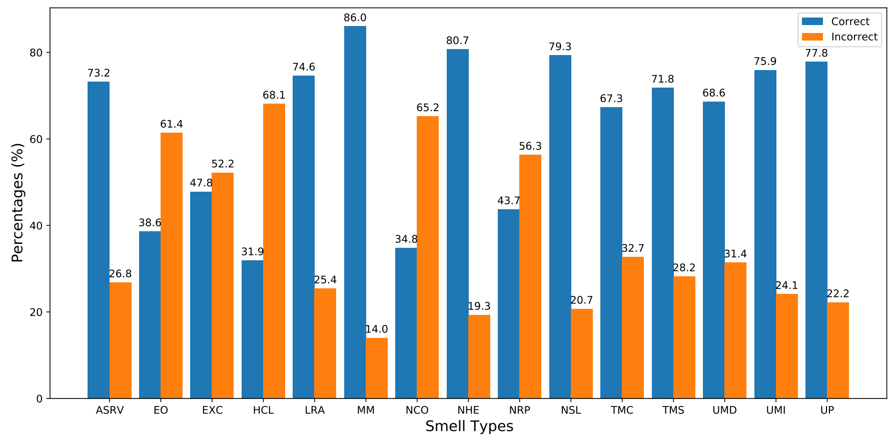
Table 4: Overall Developers' Results For the Smells Identification (Open and Closed Surveys)

Smell Names	#Correct	# Not Correct	%Correct	% Not Correct
NHE	64	19	74.95%	25.05%
UP	69	21	75.95%	24.05%
NSL	70	16	82.5%	17.5%
ASRV	69	25	73.55%	26.45%
LRA	56	19	74.8%	25.2%
NRP	36	44	49.65%	50.35%
MM	63	12	81.9%	18.1%
HCL	23	49	31.9%	68.1%
EO	22	35	38.6%	61.4%
NCO	24	45	34.8%	65.2%
TMC	56	22	74.95%	25.05%
TMS	64	25	72%	28%
EXC	44	37	66.75%	33.25%
UMD	69	22	84.3%	15.7%
UMI	50	13	87.95%	12.05%

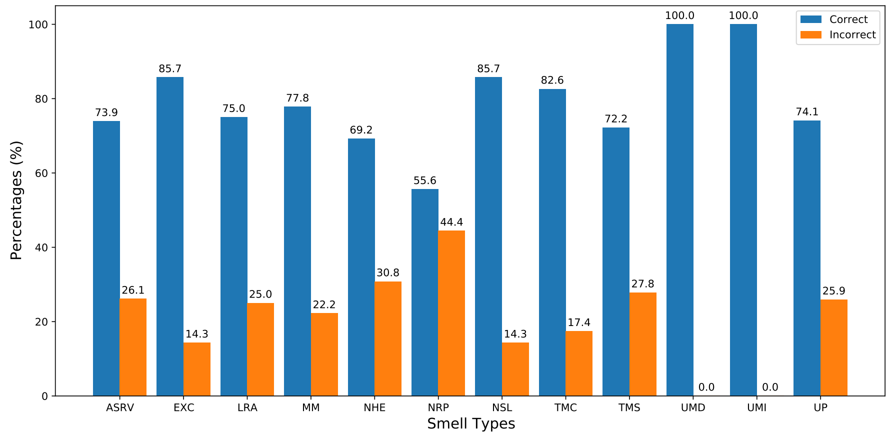
% Not Correct respectively). The percentages reported in Table 4 reflect the proportion of developers who correctly and incorrectly identified the design smells out of all the developers who answered that question. We provide in the following the developers' responses grouped in two categories, *i.e.,* smells perceived as prevalent by the majority of the participants and smells that are not perceived as prevalent by the majority of the participants.

Smells Perceived as Prevalent by the Majority of the Participants: Table 4 reports that for the design smell *Unused Method Implementation*, 87.95% (50) of the developers correctly identified the smell, while 12.05% (13) of the developers were not able to identify the design smell occurrence. For the design smell *Unused Method Declaration*, we report that 84.3% (69) developers were able to correctly identify the smell type, while 15.7% (22) were not able to identify the existing smell. For the design smells *Not Securing Libraries* and *Memory Management Mismatch*, by analyzing Table 4, we see that our results show that respectively 82.5% (70) and 81.9% (63) of the developers correctly identified these smell types. These design smell types were among the smell types correctly identified by a higher number of developers. This could be explained by the fact that most of the surveyed developers are aware of these smell types and may face them frequently [5]. As reported in Table 4, we can also observe that the design smells *Unused Parameters*, *Not Handling Exceptions*, *Too Much Clustering*, *Local References Abuse*, *Assuming Safe Return Value*, and *Too Much Scattering* were correctly identified by higher percentage of the developers.

Smells Not Perceived as Prevalent by the Majority of Participants: For the design smell *Hard Coding Libraries*, from analyzing Table 4, it appears that 31.9% (23) of the developers were able to correctly identify the smell type,



(a) General Developers



(b) Original Developers

Fig. 5: Developers' Perceived Prevalence of Multi-language Design Smells

while 68.1% (49) of the developers were not able to identify the smell. For the design smell *Excessive Objects*, we observe that 61.4% (35) of the developers were not able to identify the design smell occurrence. The design smells *Not Using Relative Path* and *Not Caching Objects* were also not correctly identified by the participants. These design smell types were among the lowest correctly identified smell types. This could be explained by the fact that these smell types are not commonly faced by the developers and that they might not be aware of these smell types. In one of our previous studies, we also observed that these types of design smells are less prevalent compared to the other smell types [6].

Our results also show that the design smells *Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, *Memory Management Mismatch*, *Unused Parameters*, *Not Handling Exceptions*, and *Too Much*

Table 5: Fisher's Exact Test Results for the Smells Identification (Open and Closed Surveys)

Smell	Exp_C	Ctrl_C	Ex_In	Ctrl_In	Odds_ratio	p_value	Con_Intr
NHE	64	23	19	49	7.176	$0.02e^{-7}$	(1.26,2.683)
UP	69	23	21	49	7.0	$0.01e^{-7}$	(1.25,2.64)
NSL	70	23	16	49	9.320	$0.03e^{-9}$	(1.5,2.97)
ASRV	69	23	25	49	5.88	$0.01e^{-6}$	(1.097,2.45)
LRA	56	23	19	49	6.278	$0.02e^{-6}$	(1.12,2.56)
NRP	36	23	44	49	1.743	0.133	(-0.11,1.22)
MM	63	23	12	49	11.185	$0.09e^{-11}$	(1.62,3.21)
EO	22	23	35	49	1.340	0.461	(-0.44,1.02)
NCO	24	23	45	49	1.136	0.725	(-0.57,0.83)
TMC	56	23	22	49	5.423	$0.02e^{-5}$	(0.99,2.39)
TMS	64	23	25	49	5.454	$0.05e^{-6}$	(1.02,2.37)
EXC	44	23	37	49	2.533	0.006	(0.27,1.59)
UMD	69	23	22	49	6.682	$0.02e^{-7}$	(1.21,2.59)
UMI	50	23	13	49	8.194	$0.04e^{-7}$	(1.32,2.89)

Clustering were identified more frequently than the others. This could be explained by the nature of these types of design smells. From analysing open source projects, we also observed that these types of design smells are among the most prevalent ones [6]. The design smells *Unused Method Implementation* and *Unused Method Declaration* are respectively related to native methods that are implemented but never called, and native methods that are declared in the Java code but never implemented in the C/C++ code. These two types of design smells could be easily identified compared to other types of smells due to their simplicity. The design smells *Not Securing Libraries*, *Memory Management Mismatch*, and *Not Handling Exceptions* are also commonly discussed in developers' blogs [1]. The issues resulting from the *Memory Management Mismatch*, and *Not Handling Exceptions* are commonly discussed in the literature [55,54]. All these reasons could explain why these types of design smells were identified more frequently than the others. Also from analyzing commit messages we found a commit message related to the design smell *Unused Parameters* in *Conscrypt* (`“Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters”`). Therefore, the prevalence of these design smell types could explain their correct identification by the developers.

To have further insights about the results presented in Table 4, we perform statistical analysis by applying Fisher's Exact test. We use this statistical test to check whether the proportion of correct identification varies between two samples (files with and without specific smell types) as discussed in Section 3. Our null hypothesis here is defined as follows. *There is no statistically significant difference between the number of developers that correctly and incorrectly identified design smells occurrences.* We use *Hard Coding Libraries* as control group, we use this smell type because it presents the lowest correct identification percentage reported by the participants. The goal here is to check whether other types of smells have a higher likelihood (Odds Ratio) of being identified by the developers. Columns **Exp_C**, **Ctrl_C**, **Ex_In**, and **Ctrl_In** contain the values of the contingency tables for the Fisher's exact test; each row corresponding to a smell type. The numbers reported in the cells of these columns are the total number of responses for correct and incorrect identification. **Ctrl_C** and **Ctrl_In** refer respectively to the number of correct and incorrect responses related to the identification of *Hard Coding Libraries*(control group). **Exp_C** and **Ctrl_In** refer respectively to the number of correct and incorrect responses related to the identification of the other smell types (experimental group).

The value of the odds ratio (OR) greater than 1 from Fisher's exact test indicates that code experiencing that design smell have higher odds of being perceived as an implementation problem compared to code without that design smell. The values $OR < 1$ indicate that code with that design smell type have lower odds of being correctly identified and perceive as an implementation problem, while $OR = 1$ refers to equal identification of design smell type. The p-value shows the probability of observing the odds ratio by chance, and thus lower values (< 0.05) of p-value confirms the significance of the impacts of design smell type on their identification. In addition to significant p-values, we examine the confidence intervals of the odds ratios. A confidence interval specifies the range where the true odds ratio lies in. To deal with the multiple testing problem, we applied the Bonferroni correction. The Bonferroni correction presents the most common way to control the family-wise error rate. For that, we computed the Bonferroni critical value by dividing the family-wise error rate (0.05) by the number of tests (in our case 14 tests). By applying the Bonferroni correction, we had 0.00357 as the Bonferroni critical value. Therefore, a significant p-value value (< 0.00357) of an odds ratio (> 1.0) with a confidence interval not containing 1 confirms a true relationship between design smell types and their likelihood of correct identification.

Fisher results support the results presented in Table 4. The results indicate a significant difference of proportions between incorrect identification of the smell type (*Hard Coding Libraries*) and other smell types. For the design smells *Not Using Relative Path*, *Not Caching Objects*, and *Excessive Objects*, Table 5 reports odds ratios close to one with confidence interval not higher than one. The obtained p-values for these design smells are not significant, not indicating a strong likelihood difference with the smell *Hard Coding Libraries*. Such results confirm our findings reported in Table 4 and indicates that the event of interest *i.e.,* correct identification is equally likely in both samples. For the other design smell types, Table 5 reports odds ratios higher than one, which indicates that the correct identification is more likely to occur in this group. From the above results, we conclude that Fisher's exact test indicates a significant difference of proportions between the correct identification of the smell types *Hard Coding Libraries*, *Not Using Relative Path*, *Not Caching Objects*, *Excessive Inter-language Communication*, and *Excessive Objects*, and the other design smell types. These smells were reported to have the lowest correct identification percentage in Table 4.

Summary of findings (RQ1): Overall, developers consider the proposed design smells to be reflective of design and implementation problems (especially the design smells: *Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, and *Memory Management Mismatch*). This provides evidence of their prevalence in multi-language systems.

4.2 RQ2: What are the reasons behind introducing multi-language design smells?

Approach To capture the possible reasons behind the introduction of the studied design smells, we asked the participants to provide their opinion about the introduction of the multi-language design smells. This question is an open question. Therefore, we performed card sorting and manual labeling to assign the responses to different categories.

Findings Since **RQ2** is based on the analysis of an open question, we performed a coding process as described in Section 3.4. For that, two authors analysed all the answers reported by the participants in order to extract general categories. We report in Figure 6 examples of the coding process and the labels assigned to the participants' responses to capture the reasons behind the introduction of multi-language design smells. Table 6 summarizes the reasons for introducing design smells along with the corresponding percentages of developers that reported the reasons. We provide in the following the definition of each category related to the reason for the introduction of multi-language design smells along with the inclusion and exclusion criteria. Note that each answer reported by the participants is assigned to only one label. Therefore, the categories presented in this section are exclusive. The percentages assigned to each category in Table 6 reflects the proportion of developers that reported that category as possible potential reasons for the introduction of design smells. The percentages are computed out of the total number of non-null answers received.

– **Refactoring and maintenance activities:** This category includes responses where the participants explicitly used the terms refactoring, restructuring, or maintenance as a possible reason for the introduction of the design smell. It includes reasons related to altering and editing the code as reported in Figure 6. This category excludes any other answers that do not mention these terms.
– **Continuous development (perform regular development tasks):** This category encompasses situations where the reasons provided by the participants were related to regular development tasks. It includes any reason provided related to coding and developments tasks except situations where they explicitly mention restructuring/refactoring/maintenance related reasons. Figure 6 provides examples of the coding process related to this category.

- **Ease of implementation:** This category contains responses where the participants related the reason for introducing design smells to the ease of coding and features integration. It includes terms that reflect simplicity, facility, easability, etc. as reported in Figure 6.
- **Lack of experience:** This category is related to situations where the participants reported a lack of development experience as possible reasons for introducing design smells. This includes responses pointing to lack of knowledge, experience, etc. as reported in Figure 6.
- **Specific implementation/design choices:** This category includes situations where the participants explicitly mentioned that code with occurrence of the design smell was the correct way of implementation.

Most of the reported reasons are related to *refactoring and maintenance activities* (37.7%) (*e.g., "When I started working on the project source code was already part of it. This is probably the product of an earlier refactoring"*), closely followed by *continuous development (perform regular development tasks)* (35.32%), this category encloses cases where the participants reported reasons that describe the development task that was performed, (*e.g., "We use it to access JMPlayer embedded functions"*). This category includes general development tasks as reported in Figure 6. Participants also pointed to situations where they described reasons related to the category *Ease of implementation* (17.06%) (*e.g., "This makes it easier to load external libraries"*). Participants also reported that this could be the results of *Lack of experience and knowledge* (5.95%) (*e.g., "it seems like it's having fun allocating memory despite being in a language where that is usually not the point"*). There were also reasons related to specific implementation and design choices (3.57%) (*e.g., "Public methods should primarily expose the logical operations"*).

Previous work investigating mono-language design smells also reported that design smells could be introduced during refactoring and maintenance activities [57]. While refactoring activities are in principle aimed at improving the design of a system, our results suggest that multi-language design smells could be introduced during refactoring activities. This could be explained by time pressure to deliver the project, which may increase the risk of bad design decisions. Another explanation could be developers who do not have a global view of the whole project, when deciding what to keep and what to remove during maintenance activities. This may result in occurrences of design smells *e.g., Unused Parameters*, *Unused Method Declaration*, *Unused Method Implementation*, and *Too Much Clustering.* Also considering the design smells *Too Much Clustering* and *Too Much Scattering*, such smells are manifested by bloated classes with an excessive number of native methods resulting in thousands of lines of code. They also manifest when developers try to isolate the native code as much as possible resulting in small class sizes with dispersed native code. Such code is hard to refactor and maintain which may lead to the introduction of the studied design smells. Our results in Table 6 show that some participants reported a lack of experience and knowledge about multi-language programming. As explained in Section 2, the combination of Java

Table 6: Participants' Perceived Reasons for the Introduction of Smells

Category Name	% Percentages	# Number
Refactoring and maintenance	37.7%	95
Continuous development(perform the task)	35.32%	89
Ease of implementation	17.06%	43
Lack of experience and knowledge	5.95%	15
Specific implementation/design choices	3.57%	9

and C/C++ requires some fundamental knowledge of the two programming languages. Mishandling some fundamental concepts (*e.g.,* types conversion, memory allocation) may lead to occurrences of design smells.

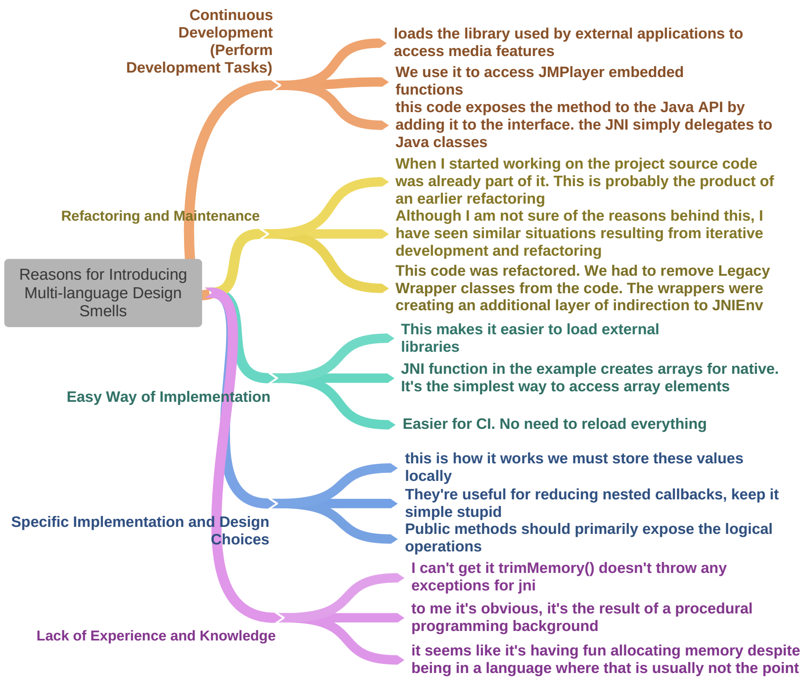

Reasons for Introducing Multi-language Design Smells
Continuous Development (Perform Development Tasks)
loads the library used by external applications to access media features
We use it to access JMPlayer embedded functions
this code exposes the method to the Java API by adding it to the interface. the JNI simply delegates to Java classes
Refactoring and Maintenance
When I started working on the project source code was already part of it. This is probably the product of an earlier refactoring
Although I am not sure of the reasons behind this, I have seen similar situations resulting from iterative development and refactoring
This code was refactored. We had to remove Legacy Wrapper classes from the code. The wrappers were creating an additional layer of indirection to JNIEnv
Easy Way of Implementation
This makes it easier to load external libraries
JNI function in the example creates arrays for native. It's the simplest way to access array elements
Easier for CI. No need to reload everything
Specific Implementation and Design Choices
this is how it works we must store these values locally
They're useful for reducing nested callbacks, keep it simple stupid
Public methods should primarily expose the logical operations
Lack of Experience and Knowledge
I can't get it trimMemory() doesn't throw any exceptions for jni
to me it's obvious, it's the result of a procedural programming background
it seems like it's having fun allocating memory despite being in a language where that is usually not the point

Fig. 6: Reasons for Introducing Multi-language Design Smells

Summary of findings (RQ2): The main reasons for introducing design smells reported by the participants are: refactoring and maintenance, continuous development (perform regular development tasks), ease of implementation, lack of knowledge, and specific implementation and design choices.

4.3 RQ3: What is the perceived impact of multi-language design smells?

Approach To assess the potential impacts of multi-language design smells, we asked the developers about their perception of the impacts of the studied multi-language design smells on selected software quality attributes. For each design smell type, we asked the developers to evaluate whether it presents a negative, neutral, or positive impact on the specified quality attributes. The developers were given the option 'N/A' in case they were not able to evaluate the impact. We computed the percentages of responses, where each participant reported that a design smell of type i has an impact (*i.e.,* positive, negative, neutral, or N/A) on the quality attribute q as described in Section 3.

Findings We report in Table 7 the summary of the perceived impact of multi-language design smells on software quality attributes, *i.e.,* expandability, modularity, reusability, understandability, performance, simplicity, and learnability. In general, the developers perceive the studied design smells as having a negative impact on the selected quality attributes. In the following, we discuss in detail the results of each design smells for the top three quality attributes (based on the percentages of the developers' responses) that were perceived to be negatively impacted by the design smell.

Too Much Clustering From our results, 99.42% of the developers perceived a negative impact on the software simplicity. 95.32% and 93.57% of the developers reported respectively a negative impact on software understandability and learnability. This design smell adds complexity to the code and may impede the understandability and learnability as its implementation results in large classes with several native methods.

Unused Method Declaration From the analysis of the results in Table 7, it appears that 94.15%, 93.57%, and 88.89% of the developers reported a negative impact on the software expandability, simplicity, and understandability respectively. This design smell introduces unused code as it is defined by native methods that are declared in Java code but never implemented in the native code. Such unused code may impede the software understandability. Indeed, it may be a challenging task for developers involved in a sub part of the system concerned by the smell, to clearly determine whether the smelly method is used by other components or not. Therefore, the expandability and reusability of components with occurrences of the design smell *Unused Method Declaration* may also be reduced.

Unused Method Implementation From our results, we can observe that 97.08% of the developers reported a negative impact on the software simplicity, 94.15% reported a negative impact on the understandability, while 92.4% of the developers reported that this design smell negatively impacts the learnability. Similar to the design smell *Unused Method Declaration*, it is expected that unused code presents a negative impact on the software understandability.

Table 7: Perceived Impacts of Multi-language Design Smells

Smells	Impact	EXP	SIM	RUS	LRN	UND	PER	MOD
NHE	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	21.05	25.73	24.56	23.98	15.2	26,90	43.86
	Negative	77.78	73.1	74.27	74.85	83.63	71,93	54.97
UP	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	13.45	8.19	14.04	36.84	8.19	19,3	16.96
	Negative	86.55	91.81	85.96	63.16	91.81	80,70	83.04
NSL	N/A	1.75	1.75	1.75	1.75	1.75	1.75	1.75
	Positive	0.0	20.47	9.36	12.28	8.77	8.77	0.0
	Neutral	39.18	14.04	22.81	28.65	39.77	32,75	37.43
	Negative	59.06	63.74	66.08	57.31	49.71	56,73	60.82
ASRV	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	8.77	0.0	0.0	0.0	0.0	0.0
	Neutral	15.79	22.22	15.2	14.04	16.37	17,54	12.28
	Negative	84.21	69.01	84.8	85.96	83.63	82,46	87.72
LRA	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	11.7	0.0	0.0	3.51	0.0	0.0
	Neutral	12.28	12.87	5.26	21.64	19.88	14,04	12.87
	Negative	86.55	74.27	93.57	77.19	75.44	84,80	85.96
NRP	N/A	1.17	1.17	1.17	1.17	1.17	1.17	1.17
	Positive	0.0	7.02	0.0	2.34	5.26	3,50	0.0
	Neutral	69.59	70.18	40.35	67.84	59.65	59,06	16.37
	Negative	29.24	21.64	58.48	28.65	33.92	36,26	82.46
MM	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	18.71	0.0	4.09	0.0	0.0	0.0
	Neutral	12.28	5.26	15.79	25.73	23.39	22.22	22.22
	Negative	87.13	75.44	83.63	69.59	76.02	77.19	77.19
HCL	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	13.45	24.56	15.79	17.54	11.11	25,15	18.13
	Negative	86.55	75.44	84.21	82.46	88.89	74,85	81.87
EO	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	8.19	2.92	11.7	13.45	9.94	13,45	9.36
	Negative	91.81	97.08	88.3	86.55	90.06	86,55	90.64
NCO	N/A	2.34	2.34	2.34	2.34	2.34	2.34	2.34
	Positive	0.0	24.56	0.0	0.0	0.0	0.0	0.0
	Neutral	21.64	5.26	10.53	35.67	20.47	22.22	35.67
	Negative	76.02	67.84	87.13	61.99	77.19	75,44	61.99
TMC	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	10.53	0.0	9.36	5.85	4.09	8,19	25.15
	Negative	88.89	99.42	90.06	93.57	95.32	91,23	74.27
TMS	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	7.02	14.04	7.02	12.28	8.77	14,62	13.45
	Negative	92.98	85.96	92.98	87.72	91.23	85,38	86.55
EXC	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	14.62	9.36	12.87	9.36	6.43	13,45	8.77
	Negative	85.38	90.64	87.13	90.64	93.57	86,55	91.23
UMD	N/A	0.58	0.58	0.58	0.58	0.58	0.58	0.58
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	5.26	5.85	13.45	12.28	10.53	12,28	33.92
	Negative	94.15	93.57	85.96	87.13	88.89	87,13	65.5
UMI	N/A	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Positive	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Neutral	14.04	2.92	17.54	7.6	5.85	9,36	8.19
	Negative	85.96	97.08	82.46	92.4	94.15	90,64	91.81

EXP: Expandability, **SIM:** Simplicity, **RUS:** Reusablity, **LRN:** Learnability
UND: Understandability, **MOD:** Modularity, **PER**: Performance
NHE: NotHandlingExceptions, **ASRV:** AssumingSafeReturnValue
UP: UnusedParameters, **UMI:** UnusedMethodImplementation
UMD: UnusedMethodDeclaration, **NRP:** NotUsingRelativePath
NSL: NotSecuringLibraries, **EXC:** ExcessiveInterlangCommunication
NCO: CachingObjects, **MM:** MemoryManagementMismatch
LRA: LocalReferencesAbuse, **HCL:** HardCodingLibraries
TMS: ToomuchScattering, **TMC:** Toomuchclustring, **EO:** ExcessiveObjects

Given the characteristics of this design smell, it is not surprising that it is considered to have negative impacts on the software's simplicity and learnability.

Unused Parameters Table 7 shows that 91.81% of the developers reported a negative impact of this design smell on both software understandability and simplicity. 86.55% of the developers reported a negative impact on the software expandability. This design smell captures the fact that at least one of the native method parameters is not used within its native implementation. Therefore, such unused parameters may introduce some confusion and thus have negative impacts on the software understandability and simplicity.

Assuming Safe Return Value This design smell is reported by 87.72% and 85.96% of the developers, as having a negative impact respectively on the software modularity and learnability. 84.8% of the developers reported a negative impact on the software reusability. The perceived negative impacts could be related to the nature of this design smell. The modularity of the system is the degree to which the implementation of the functions is independent of one another. Therefore, not properly checking native return values may affect the software modularity and its reusability.

Memory Management Mismatch Table 7 reports that 87.13% and 83.63% of the developers reported a negative impact of this design smell on the software expandability and reusability, respectively. Our results also show that 77.19% of the developers perceive a negative impact of this design smell on the software modularity and its performance. This perception of the impact of this design smell could be explained by the purpose of the design smell. The JVM offers a set of predefined methods that could be used to access fields, methods, and convert types from Java to the native code. Those methods allocate memory to each Java object that is used within the native code. Since Java's garbage collection system has no control over the use of dynamic memory in the native code. Therefore, developers' should release the memory allocated to each Java object. Not releasing the memory may introduce bugs and security issues [55,54]. Such issues may affect the component expandability and reusability.

Local References Abuse Table 7 shows that 93.57% of the developers reported a negative impact on the software reusability, while 86.55% and 85.96% of the developers reported a negative impact on the software expandability and modularity, respectively. Similarly, to the design smell *Memory Management Mismatch*, this design smell type also may lead to memory issues [3]. JNI creates locale references for any Java object that is used in the native code. Not releasing local references may lead to memory issues, especially since the JVM enables to create a maximum of 16 local references. Exceeding the maximum number without informing the JVM may introduce bugs and memory leaks [55]. A component that is often subject to memory issues may have low reusability and expandability. Especially for JNI systems, several studies in the literature discussed the issues and vulnerabilities that could result from not properly releasing the memory [31,54].

Not Handling Exceptions This design smell was perceived by 83.63% and 77.8% of the developers to have a negative impact on the software understandability and expandability, respectively. Our results show that 74.85% of the developers reported a negative impact on the software learnability. The management of exceptions has been discussed in the literature as one of the main concerns in the context of JNI systems [54]. Indeed, occurrences of this design smell may introduce some challenges and even bugs, which could explain the negative impacts reported by the developers. Since multi-language code requires access from the host code components that are implemented in another programming language, mishandling exceptions may require additional effort to properly locate and fix the bug. Therefore, such design smell may negatively impact the understandability and learnability.

Excessive Inter-language Communication This design smell was reported by respectively 93.57% and 91.23% of the developers to negatively impact the understandability and modularity of the system. The results also show that 90.64% of the developers reported negative impacts of this design smell on the software learnability and simplicity. This design smell captures excessive communications between components written in different programming languages. Such excessive communications may impede the software understandabiliy, learnability, and simplicity.

Excessive Objects Table 7 reports that 97.08% and 91.81% of the developers reported a negative impact on the software simplicity and expandability, while 90.65% of developers reported a negative impact on the software modularity. It is not surprising that this design smell is perceived as negatively impacting the software quality, since it occurs when developers pass a whole object from Java to native code, when only some of the object fields are needed. The access to the Java object from the native code requires a call to specific methods following a specific order. Such methods may impact the simplicity and expandability of the software component.

Too Much Scattering Table 7 shows that 92.98% of the developers reported a negative impact on the software expandability and reusability, and 91.23% of the developers reported a negative impact on the understandability. The negative impacts could be explained by the fact that the smell occurs when the native declaration methods are scattered in different components of the Java code. This dispersion may impede the reusability of parts of the code and make it hard for developers to expand the code of the system.

From our results, we also observe that some quality attributes are less impacted than the others. Table 7 shows that 70.18% of the developers reported that the design smell *Not Using Relative Path* has a neutral impact on the software simplicity. 69.59% and 67.84% of the developers reported a neutral impact on the software expandability and learnability. Given the characteristics of this design smell, having a neutral impact on the expandabiliy is rather surprising as it may affect the access to the native library. However, the neutral impacts on the software simplicity and learnability could be explained by the definition of the design smell itself. Indeed, occurrences of the design smell *Not Using Relative Path* could result in a piece of code that is relatively simpler compared to its refactored solution. However, this design smell may affect the reusability of the code and increase the cost of maintenance activities if the library is no longer available.

The design smell *Not Securing Libraries* was also reported by some developers to have a neutral impact on some software quality attributes. 39.77% and 39.18% of the developers reported a neutral impact on the software understandability and expandability. Our results also show that 66.08% of the developers reported a negative impact of this design smell on the software reusability. The negative impacts of this design smells on software reusability could be expected. Indeed, insecure code leaves breaches in the code; opening it to malicious attacks, and thus could reduce the resuability of that component. The neutral impacts reported on the software understandability and expandability could be explained by the fact that this design smell results in a simple block to load the library. Occurrences of this design smell are manifested when the native library is loaded without a call to any specific methods that ensure that the library could not be loaded by unauthorized code. To refactor this design smell, developers are required to add an additional security block, which may impede the effort required to understand the code.

The results in Table 7 show that in general the design smells are perceived to have a negative impact on the studied quality attributes. Understandability of the software was reported to be negatively impacted by the design smells. Understandability is one of the main concerns during maintenance activities. Several studies in the literature discussed the challenges of understanding and maintaining multi-language systems [35, 28, 39]. We believe that occurrences of design smells are likely to increase the effort needed to understand and maintain multi-language systems. From Table 7, we observe that the design smells *Too Much Clustering* (95.32%), *Unused Method Implementation* (94.15%), and *Excessive Inter-language Communication*(93.57%) are among the design smells mostly perceived to have a negative impact on the software understandability. Reusability is also reported to be negatively impacted by the design smells. The design smells *Too Much Scattering* (92.98%) and *Too Much Clustering* (90.06) are among design smells that are perceived to have the most negative impact on the software reusability. This may be the consequence of the negative impacts of the design smells on the software quality, *i.e.,* fault-proneness as reported in our previous study [6]. *Too Much Clustering* smell occurs when too many native methods are declared in a single class creating a blob class resulting in poor comprehension and maintainability of the code. This is likely to add more risk of bugs as reflected in our previous study [6]. *Too Much Scattering* smell occurs when native methods scatters (often duplicated) over scarcely used classes which are likely to introduce maintenance challenges and increase the risk of bugs. Design smells may impact components' lifetime, limit their portability and impede their reusability. Developers considered a neutral impact on the studied quality attributes for some specific types of design smells *Not Using Relative Path*, *Not Securing Libraries*. These design smells were perceived to have a neutral impact on the simplicity. The simplicity was also reported to be negatively impacted mainly by *Too Much Clustering* (99.42%) and *Unused Method Implementation* (97.08%). This could be explained by the nature of these design smells, as they are adding extra complexity to the code and thus may affect its simplicity.

Summary of findings (RQ3): Most of the studied design smells were perceived as negatively impacting the studied software quality attributes. The design smell *Not Using Relative Path* was perceived to have a neutral impact. Having knowledge of their existence and the potential impact could help to improve the quality of multi-language systems.

4.4 **RQ4: What are the design smells that developers perceive as the most harmful?**

Approach To assess the perceived severity of multi-language design smells, we asked the developers to rank the design smells based on their perceived level of harmfulness (from the most harmful to the less harmful) using a score from 15 to 1 (15 for the most harmful, 1 for the less harmful). We asked the developers to consider the impact of the design smells on software quality during the ranking. We used the Borda Count technique to rank the candidates *i.e.,* design smells as described in Section 3. By considering the number of votes associated with each design smell, the Borda count yields a consensus-based ranking instead of a majority-based one [12].

Findings Table 8 reports the aggregated results of the Borda Count. Table 8 also provides the median perceived severity associated with each smell type. We found that *Not Handling Exceptions* is perceived as the most harmful design smells with a score of 2261, closely followed by *Assuming Safe Return Value* with a score of 2137. We also report the median severity associated to those smell types. Both *Not Handling Exceptions* and *Assuming Safe Return Value* received the highest median severity value of 12. The design smell *Local References Abuse* is also reported to be harmful with a median severity of 11, but with a Borda Count of 2063. The developers also considered the smell *Memory Management Mismatch* to be harmful with a median severity of nine and a Borda score of 2052. The design smells *Excessive Inter-language Communication* and *Too Much Clustering* were also perceived by the developers as harmful with a median severity of 11 and 10 respectively, and a Borda Count score of 2040 and 1876, respectively.

The design smells *Hard Coding Libraries*, *Not Using Relative Path*, *Unused Method Declaration*, and *Unused Parameters* were reported to be less harmful compared to the other smells with a median severity value of five for all of them. Their Borda Count scores are 746, 632, 588, and 438, respectively.

Table 8: Ranking of Multi-language Design Smells from Developers' Perception

Design Smells	Score (Borda)	Median Severity
Not Handling Exceptions	2261	12
Assuming Safe Return Value	2137	12
Local References Abuse	2063	11
Memory Management Mismatch	2052	9
Excessive Inter-language Communication	2040	11
Too Much Clustering	1876	10
Not Securing Libraries	1358	7
Too Much Scattering	1342	7
Excessive Objects	1211	6
Unused Method Implementation	964	5
Not Caching Objects	812	6
Hard Coding Libraries	746	5
Not Using Relative Path	632	5
Unused Method Declaration	588	5
Unused Parameters	438	5

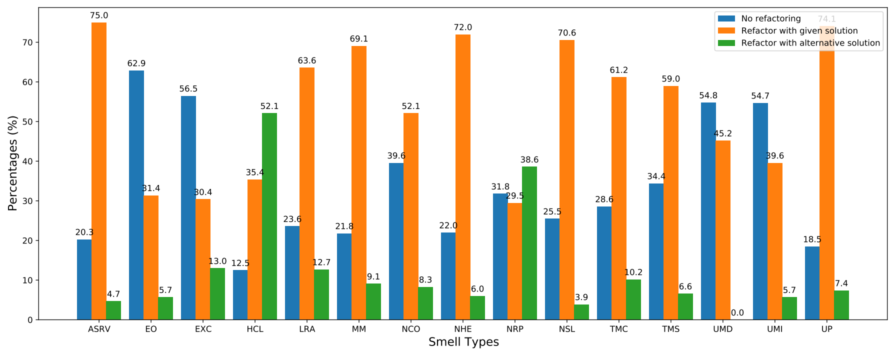
These ranking results could be explained by the definition and the nature of the design smells. It is expected to consider *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, and *Memory Management Mismatch* among the most harmful design smells. Indeed, these types of design smells could be directly related to the introduction of bugs. Some of these design smells were indeed related to bugs, *e.g., Conscrypt*: `‘‘Fixed various memory leaks and Java 8 JNI Compatibility WARNING in native method: JNI call made without checking exceptions when required to from CallObjectMethod’’`. Several studies in the literature also discussed the bugs and issues that could result from mishandling exceptions and memory issues [54, 55]. The developers reported that they perceive the design smells *Unused Method Declaration* and *Unused Parameters* as less harmful than the others. However, we believe that all the design smell types should be considered with caution, including those that are perceived to be less harmful. In fact, even if the nature of some design smells may not seem directly related to bugs, still these design smells can increase the maintenance effort and even lead to bugs, *e.g.,* `‘‘There were a bunch of exceptions that are being thrown from JNI methods that aren’t currently declared’’`, and `‘‘Fix latent bug in unused method’’` present examples of commit messages extracted respectively from *Conscrypt* and *Pljava*.

Summary of findings (RQ4): The design smells perceived to be the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering*. The design smells perceived by the developers as less harmful are: *Unused Parameters*,*Unused Method Declaration*, *Not Using Relative Path*, and *Hard Coding Libraries*. Our results highlight the importance of prioritizing multi-language smells for maintenance and refactoring activities.

4.5 **RQ5: Do developers plan to refactor those design smells?**

Approach One way to ensure the software quality is the identification and refactoring of design smells occurrences. Therefore, we aim to investigate whether developers would consider refactoring the design smell occurrences, and whether the refactoring considerations vary from one specific type of smells to the other. For each code snippet in which the developers reported a design or implementation problem, we also asked them if they would consider applying any refactoring to remove the identified problem. Specifically, we proposed a solution and asked them whether they would refactor with the given solution, refactor with an alternative solution, or whether they would not refactor. We then computed for each smell type, the percentage of developers who reported that they would refactor the code with the proposed solution, refactor the code with an alternative solution, or would not refactor the code. We also aggregate the responses to provide a general overview of the developers' opinions about refactoring multi-language design smells.

Findings Figure 7a and Figure 7b present the percentage of developers (for the open and closed surveys, respectively) who (1) selected option No, when we asked them if they would consider refactoring the design smell presented in the code snippet; (2) selected "Yes" for the given solution; or (3) selected "Yes" for any alternative refactoring solution. We also report in Table 9 an overview of the developers responses for the combination of both open and closed surveys. This table reports for each smell type, #**No** and **%No**: the total number and percentage of developers that reported that they would not consider refactoring the design smells. #**Yes_Gi** and **%Yes_Gi**: the total number and percentage of developers that reported that they would apply the proposed solution. #**Yes_Alt** and **%Yes_Alt**: the total number and percentage of developers who reported that they would refactor with an alternative solution. The percentages values reported in Table 9 reflect the proportion of developers who reported that they would not refactor, refactor with a given solution, refactor with an alternative solution out of all the developers who answered that question. From our results in Table 9, we observe that developers' decision on refactoring is dependent on the design smells types.


(a) General Developers

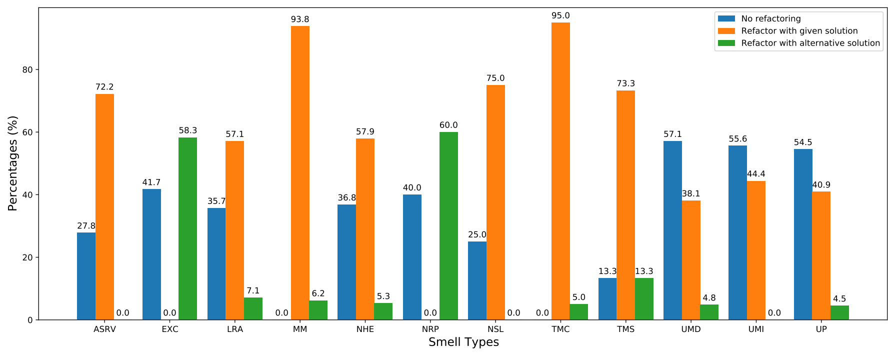

(b) Original Developers

Fig. 7: Developers' Perceived Refactoring Considerations for the Design Smells

Smells that would be refactored: From the results presented in Table 9, we can observe that for the design smell *Memory Management Mismatch*, 81.45% out of all the developers who answered this question reported that they would apply the proposed refactored solution while only 10.9% reported that they would not consider refactoring that design smell. For the design smell *Too Much Clustering*, 78.1% of the surveyed developers reported that they would consider the proposed refactoring solution, and 14.3% would not consider refactoring that design smell. For the design smells *Assuming Safe Return Value* and *Not Securing Libraries*, 73.6% and 72.8% of the developers respectively reported that they would consider the proposed solution for refactoring. These results confirm and complement our earlier empirical evidence reported from analysing open source projects and the commit messages [6]. Indeed, we have reported that some of the studied design smells were explicitly removed by the developers due to possible improvements or bugs these smell types could introduce (*e.g.,* in *Realm*, a commit message was discussing errors related to memory management ‘‘`DeleteLocalRef when the ref is created`

Table 9: Overview Developers' Results For Refactoring the Smells (Open and Closed Surveys)

Smell	#No	#Yes_Gi	#Yes_Alt	%No	%Yes_Gi	%Yes_Alt
NHE	18	47	4	29.4	64.95	5.65
UP	22	49	5	36.5	57.5	5.95
NSL	19	54	2	25.25	72.8	1.95
ASRV	18	61	3	24.05	73.6	2.35
LRA	18	43	8	29.65	60.35	9.9
NRP	16	13	20	35.9	14.75	49.3
MM	12	53	6	10.9	81.45	7.65
HCL	6	17	25	12.5	35.4	52.1
EO	22	11	2	62.9	31.4	5.7
NCO	19	25	4	39.6	52.1	8.3
TMC	14	49	6	14.3	78.1	7.6
TMS	23	47	6	23.85	66.15	9.95
EXC	31	14	13	49.1	15.2	35.65
UMD	46	36	1	55.95	41.65	2.4
UMI	34	25	3	55.15	42	2.85

`in loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it''`).

Smells that would not be refactored: When asking the developers if they would consider refactoring the occurrences of the design smell *Excessive Objects*, 62.9% of the developers selected the option "No", meaning that they would not refactor the code to remove the design smell. The results show that 31% reported that they would consider applying the given refactoring solution, while only 5.7% reported that they would use an alternative refactoring solution to remove the smell. This result could be explained by safety concerns. When features are mixed together with excessive calls between components written in different programming languages, a change to the behavior of one may cause a bug in another feature. Therefore, developers may be reluctant to remove the design smell to avoid possible side effects that can cause bugs. For the design smells *Unused Method Implementation* and *Unused Method Declaration*, 55.95% and 55.15% of the surveyed developers respectively reported that they would not consider refactoring the design smell, while 41.65% and 42% respectively reported that they would apply the proposed refactoring solution. These results could also be related to safety concerns. Native implementations and methods could be tricky to refactor since they are declared in one programming language and could be used in another programming language as described in Section 2. Thus, refactoring such occurrences may introduce bugs if the methods are still used by external code.

From Figure 7a and Figure 7b, we observe that there are a few cases where the choice about the refactoring varies between the open and closed surveys. For example, for the design smell *Unused Parameters*, while in the open survey 75% of the developers reported that they would consider applying the proposed refactoring, 54.5% of the original developers (in the closed survey) reported

Table 10: Fisher's Exact Test Results for the Smells Refactoring (Open and Closed Surveys)

Smell	Ex_Y	Ctrl_Y	Ex_N	Ctrl_N	Odds_ratio	p_value	Con_Inter
NHE	51	13	18	22	4.7955	$0.05e^{-3}$	(0.7,2.44)
UP	54	13	22	22	4.154	$0.09e^{-3}$	(0.578,2.27)
NSL	56	13	19	22	4.988	$0.36e^{-3}$	(0.75,2.47)
ASRV	64	13	18	22	6.017	$0.04e^{-4}$	(0.93,2.66)
LRA	51	13	18	22	4.795	$0.05e^{-3}$	(0.7,2.44)
NRP	33	13	16	22	3.490	0.008	(0.34,2.16)
MM	59	13	12	22	8.3205	$0.04e^{-5}$	(1.194,3.04)
HCL	42	13	6	22	11.846	$0.04e^{-5}$	(1.38,3.57)
NCO	29	13	19	22	2.583	0.046	(0.05,1.85)
TMC	55	13	14	22	6.648	$0.27e^{-4}$	(0.99,2.8)
TMS	53	13	23	22	3.899	0.002	(0.52,2.20)
EXC	27	13	31	22	1.474	0.396	(-0.47,1.25)
UMD	37	13	46	22	1.361	0.542	(-0.50,1.12)
UMI	28	13	34	22	1.394	0.523	(-0.52,1.18)

that they would not consider refactoring that design smell. The same goes for the design smell *Excessive Inter-language Communication*; while 56.5% of the developers of the open survey reported that they do not consider refactoring the design smell, 58.3% of the original developers (in the closed survey) reported that they would consider refactoring the design smell with an alternative solution.

From Figure 7a and Figure 7b, we observe that in general, almost all the developers reported that they would consider refactoring the design smells. This may indicate that multi-language design smells are perceived by the developers as harmful, and refactoring them is likely to be among their priorities. This could be resulting from the impacts of the design smells on software quality. As reported in the two previous research questions, the developers perceive the design smells as harmful and consider them to have a negative impact on software quality. In addition, in most of the situations, the developers reported that they would consider applying the refactoring solutions that we proposed. Therefore, we believe that a refactoring approach could be considered to improve the quality of multi-language systems by removing the occurrences of multi-language design smells. From analyzing bug-fixing commit messages, we identified some commits reporting a refactoring for specific smells, *e.g.,* `“removing unused parameter”`, `“implementing the handling of exception”`. Those commit messages suggest that developers often refactor occurrences of the studied design smells.

Similar to **RQ1**, to have further insights into the results of **RQ5**, we performed statistical analysis by applying Fisher's Exact Test as described in Section 3. Our null hypothesis here is defined as follows. *There is no statistically significant difference between the percentages of developers considered to refactor and not to refactor the design smells occurrences.* We use *Excessive Objects* as control group. This design smell presents the highest percentage of participants choosing not to refactor. The goal here is to check whether other types of smells have a higher likelihood (Odds ratio) of being refactored by the developers. Table 10 reports the results of applying Fisher's Exact test. Table

10 reports the values of the contingency tables for the Fisher's exact test; each row corresponding to a smell type. The numbers reported in the cells of these columns are the total number of responses for which participants reported that they would consider to refactor or not to refactor that design smell type (refactoring results for Yes_Given and Yes_alternative solutions are aggregated). **Ctrl_Y** and **Ctrl_N** refer respectively to the number of developers who reported that they would and would not refactor the design smell *Excessive Objects* (control group). **Exp_Y** and **Ctrl_N** refer respectively to the number of developers who reported that they would and would not refactor the other smell types (experimental group). Fisher's test results support the results presented in Table 9. For almost all the design smell types, Table 10 reports odds ratios higher than one, with significant p-values. To deal with the multiple testing problem, similar to **RQ1**, we applied the Bonferroni correction. By applying the Bonferroni correction, we had 0.00357 as the Bonferroni critical value. Therefore, a significant p-value value < 0.00357 (corrected as $0.05/14$) of an odds ratio (> 1.0) with a confidence interval not containing 1 confirms a true relationship between design smell types and their likelihood of refactoring decision. The results indicate a significant difference of proportions between the refactoring decision of the smell type (*Excessive Objects*) and other design smell types. For the design smells *Excessive Inter-language Communication*, *Unused Method Declaration*, and *Unused Method Implementation*, Table 10 reports odds ratios close to one with confidence interval not higher than one and with non-significant p-values. Such results confirm our findings reported in Table 9 and indicate that the event of interest, *e.g.,* refactoring decision is equally likely in both samples, *i.e.,* experimental group, and control group as described in Section 3. Therefore, we conclude that most of the participants as shown in Table 9 reported that they would not consider refactoring the design smell *Excessive Objects*, *Unused Method Declaration*, and *Unused Method Implementation*, while they would consider refactoring most of the other design smell types and this difference is statistically significant (except for *Not Using Relative Path* and *Not Caching Objects*). Our results suggest that specific refactoring tools could help to improve the quality of multi-language systems.

Summary of findings (RQ5): The refactoring consideration varies from one specific smell type to another. However, for the majority of smell types, the developers reported that they would consider refactoring the design smells. This might be indicative of the developers' concern related to the studied design smells and refactoring of those likely to improve the quality of multi-language systems.

5 Discussion and Implications

We now discuss the results reported in Section 4.

5.1 Developers' Perception of Multi-language Design Smells

Prevalence of Multi-language Design Smells: From our results in **RQ1**, we observe that developers were mainly able to correctly identify the following design smells: *Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, *Memory Management Mismatch*, *Unused Parameters*, *Not Handling Exceptions*, and *Too Much Clustering*. The design smell *Unused Method Implementation* is defined by a native method that is implemented but never called from the host code, while the design smell *Unused Method Declaration* is about a native method that is declared in the host code but is never implemented in the native code. In the same vein, the design smell *Unused Parameters* is about a parameter that is passed from the java to the native code without being part of the native implementation. These design smells are related to unused code. Therefore, based on our previous research studies and experience with the analysis of multi-language systems [4,3,2,6], we believe that it is not very surprising that the developers were able to identify them easily, which provides additional evidence of their prevalence and existence in multi-language systems. Since multi-language systems are emerging from the concept of combining components written in different languages and they generally involve different developers who might not be part of the same team. It could also be a challenging task for a developer working only on a sub-part of a project to clearly determine whether that specific parameter or method is used by other components.

Regarding the design smell *Too Much Clustering* it is also not surprising to see that developers were able to identify it correctly, because we believe that this type of design smell is widespread; developers may frequently observe occurrences of this design smell type [6]. The design smells *Memory Management Mismatch* and *Not Handling Exceptions* are commonly discussed in several research articles and developers' blogs discussed bugs related to mishandling JNI exceptions and the management of the memory [55,54, 3]. Therefore, it is also expected that most of the developers were able to identify these types of design smells. Similarly, another commit message was reporting on the design smells *Not Handling Exceptions* and *Memory Management Mismatch* in *Conscrypt* (`“rework exceptions throwing from jni”`, `“Added error handling of all uses of sk_*_push which can fail due to out of memory”`). On the other hand, most of the developers were not able to correctly identify the design smells *Hard Coding Libraries*, *Excessive Objects*, *Not Using Relative Path*, and *Not Caching Objects*. From our results, most of the responses related to the identification of these design smell types are resulting from the open survey as shown in Table 3. Therefore, the perceived prevalence for these design smell types is resulting from the general developers and not the original developers. For these design smell types, our results reflect the perceived prevalence from the perspective of general developers.

Impacts of Multi-language Design Smells on Software Quality: From analyzing the results of **RQ4**, we found that the developers reported design smells *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering* as the most harmful types of design smell. While the design smells *Unused Parameters*, *Unused Method Declaration*, *Not Using Relative Path*, and *Hard Coding Libraries* are perceived as less harmful.

Considering the design smells *Not Handling Exceptions* and *Assuming Safe Return Value* as harmful could be explained by the possible issues that could arise from these types of design smells. Indeed, the management of exceptions may not be automatically ensured depending on the programming languages. In some situations, developers should explicitly implement the exception handling flow. Similarly, return values are often used in multi-language systems to pass objects and values from one language to the other. Thus, it is important to ensure that the interaction between components of different programming languages is successfully completed. Otherwise, bugs and issues may occur [3,2,55,54]. The design smells *Local References Abuse* and *Memory Management Mismatch* were both reported as harmful. These design smells may lead to memory leaks which are also commonly discussed in the JNI development literature [3,2,55,54]. The management of the memory should be considered separately for both Java and C/C++ sides. Unlike Java, the C language requires developers to explicitly take care of the management of the memory using available functions such as `malloc` and `free`. However, this leak and mismatch between Java and C/C++ is a source of programming defects since it is introducing security vulnerabilities. The manual process of managing the memory is likely to introduce memory leaks. Thus, developers should pay more attention to memory management when they are in the context of multi-language development. An example of commit message in *Realm* reflects a case of bug related to the design smell *Local References Abuse* (`‘‘Local ref needs to be cleaned on client thread (#4830), Clean the local ref after notifyAllChanges Downloaded’’)`. In *Rocksdb*, we also found a commit message describing a bug resulting from the design smell *Memory Management Mismatch* (`‘‘As raised in #2265, the arena allocator will return memory that is improperly aligned to store a ‘std::function‘ on macOS.’’)`

Introduction of Multi-language Design Smells: Our results from **RQ2** emphasize that the main reasons for introducing design smells are related to refactoring and maintenance activities, and the continuous development (perform regular development tasks). This could be resulting from the fact that similar to mono-language systems, for multi-language systems also, occurrences of design smells could be introduced due to time pressure. Results also suggest that another reason for the introduction of the design smell is related to implementation complexity vs quality trade-off, leading to the developer's choice of the ease of implementation due to time constraints. This finding could be explained by the fact that some of the design smells types are defined by a missing check or a call to specific methods. Therefore, even if the implementation would look simpler without the occurrence of some types of the design smells, *e.g., Not Handling Exceptions*, *Memory Management*

Mismatch, *Local References Abuse*, *Assuming Safe Return Value*, these design smells increase the maintenance activities and could introduce bugs [6]. Hence, we believe that developers should be cautious with the occurrences of the studied design smells. Our results also report situations in which the participants explicitly reported a lack of experience and knowledge. This finding could be explained by the lack of established guidelines and common practices that developers should consider when dealing with multi-language systems. Also, those systems require developers working on any of the system's components to have experience in multiple programming languages. In addition, developers are required to consider the synchronisation and the data conversion between different programming languages since each programming language has its own rules (*i.e.,* lexical, semantic, and syntactical). Therefore, we believe that formal guidelines could help improve the quality of multi-language systems and reduce their challenges by reducing the occurrences of design smells. Since our results report that the studied design smells could be introduced while performing daily development activities, *e.g.,* refactoring, and continuous development. Having knowledge of the occurrences of design smells could improve the quality of multi-language systems.

Refactoring of Multi-language Design Smells: From our results in **RQ5**, we found that in general, the developers would consider refactoring design smells occurrences. Our results also suggest that the developers would apply the proposed solution for refactoring. Hence, we believe that an approach to automatically remove such occurrences of smells has a good chance of adoption by developers. During our analysis, we also observed situations where the original developers of a smelly file reported that they would not consider refactoring the occurrence of the design smells. This is the case for the design smell *Unused Parameters*, for example. We attribute these developers' decisions to the risk of side effects that could result from refactoring parts of the code on which they have imperfect knowledge. We also attribute these decisions in part to the nature of multi-language programming. Since multi-language systems may involve different teams who contribute separately using different programming languages, developers' may not have a global view of the whole system to decide whether for example a parameter could be removed without causing bugs or breaking changes to other related components (*e.g.,* in the case of the *Unused Parameters*design smell). However, as presented in some examples of extracted commit messages, there were situations in which the developers explicitly reported removing occurrences of the design smell *Unused Parameters* because there were bugs resulting from this design smell type. Therefore, we believe that developers should consider refactoring occurrences of multi-language design smells whenever possible.

5.2 Comparative Insights Regarding Previous Works

In a previous work [6], we studied the prevalence of design smells in open source projects. Our results show that most of the design smell types are prevalent in open source projects, in particular *Unused Parameters*, *Too Much*

Scattering, *Unused Method Declaration*, while others are less prevalent, *e.g.,* *Excessive Objects* and *Not Caching Objects*. The results from surveying developers confirm this finding of the prevalence of the design smells. Therefore, we believe that most of the design smells studied in this paper are prevalent and are considered by developers to reflect design and implementation problems.

Regarding the severity of multi-language design smells, in our previous study [6], we reported that files with occurrences of multi-language design smells are more likely to be subject to bugs than files without those types of design smells. We also report that some specific types of design smells are more related to bugs than others: *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding Libraries*, *Not Handling Exceptions*, *Memory Management Mismatch* and *Not Securing Libraries*. From surveying developers, we found that in general, most of the design smells have a negative impact on software quality attributes, and that they are considered harmful by developers. However, some specific types are reported to be more harmful than the others: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, *Excessive Inter-language Communication*, and *Too Much Clustering* are reported to be the most harmful design smell types. The design smells *Hard Coding Libraries*, *Not Using Relative Path*, *Unused Method Declaration*, and *Unused Parameters* were reported by the developers as less harmful than the other types of smells. Comparing the findings from our previous study and this study, we observe that while some types of design smells are considered by developers as less harmful, they are among of the most harmful design smells and were reported to increase the risk of introducing of bugs [6]. Indeed, developers reported that the design smells *Unused Parameters* and *Hard Coding Libraries* are less harmful compared to other types of design smells. However, from analyzing open source projects, we found that these two types of design smells are considered among the most bug-prone types of design smells. Studying each type of smell separately also allowed us to capture their impact individually. Surveying developers about the harmfulness and perceived impacts of each smell type separately also allowed us to capture the developers' perception versus the real impacts in open source projects. The insights from this study could help developers to prioritize multi-language smells for maintenance and refactoring activities. Especially that some of the design smells that are not perceived as prevalent do actually have an impact on the software fault-proneness as reported in our previous empirical study [6]. We have observed commit messages indicating refactoring for removing specific smells that caused bugs (*e.g.,* commit messages: *e.g.,* `‘‘There were a bunch of exceptions that are being thrown from JNI methods that aren't currently declared’’`, `‘‘cleaning up JNI exceptions (#252)’’`, `‘‘removed a few unused JNI methods’’` extracted respectively from *Conscrypt* and *Pljava*). Therefore, we believe that the smell types *Unused Parameters*, *Too Much Clustering*, *Too Much Scattering*, *Hard Coding Libraries*, *Not Handling Exceptions*, *Memory Management Mismatch* and *Not Securing Libraries*. should be considered in priority since their occurrence seems to increase the risk of bug introduction. Both developers' perception and the empirical evidence presented in our previous study [6] are important regarding the impacts of the smells on the quality of multi-language systems. Although both studies provide us with important insights about the impacts of multi-language design smells, we believe that further research is required to have more conclusive evidence.

We believe that the findings of this paper provide insights for the research community in general, but also for developers, and any of those considering the use of multi-language programming. Our results report that most of the studied design smells are relevant *i.e., Unused Method Implementation*, *Unused Method Declaration*, *Not Securing Libraries*, *Memory Management Mismatch*, *Unused Parameters*, *Not Handling Exceptions*, and *Too Much Clustering*. We also reported that the design smells present a negative impact on the software quality attributes and that they are considered as harmful. Therefore, we believe that developers should pay attention to the studied design smells. Our results emphasize that in many situations, the participants reported that they could consider refactoring occurrence of the studied design smells. Hence, we believe that refactoring approaches should be considered to automatically remove the occurrences of multi-language design smells.

6 Threats To Validity

In this section, we discuss the threats to the validity of our study [61].

Threats to Construct Validity These threats concern the relationship between theory and observation. According to Fink, a survey allows to collect information from or about people to describe, compare, or explain their knowledge and behavior about a specific topic [13]. We followed guidelines in the coding process of the open questions as described in section 3. Two of the authors independently analysed the developers' responses and performed the coding process. The initial inter-rater agreement rate was 94.84%. We then resolved the disagreements through discussions with the research team. Concerning the measure of perception, we asked developers to report to us whether they perceived a problem in the code shown to them. In addition, we asked them to specify the smell name in order to understand whether or not they were able to correctly identify the design smells studied. We are aware that surveys only reflect a subjective perception of the problem, and might not fully capture the extent to which the design smells could be relevant or harmful. Regarding the sample of smells used in this survey, to mitigate any possible threats related to the recall and-or precision of the detection approach, we manually validated all the instances of design smells that were used in the survey. The validation process was based on the definition of the smells and their related rules. We validated that the code snippets presented to the developers follow the rules introduced when defining and documenting the design smells [4,3,6]. Even if not all the surveyed developers correctly identified the design smells presented in this study, we still believe that these are concrete design smells and could have negative impacts on the source code as reported in our previous study

[6]. Our study is an internal validation of multi-language design smells that we previously defined and cataloged. Thus, this may present a threat to validity. However, this threat was mitigated by publishing our catalog in a pattern conference. The catalog of design smells went through several rounds of validation and most of the design smells presented in this paper were discussed and-or reported by multi-language developers [4,3,41]. However, as future work, we plan to contrast the developers' perception and the empirical results obtained from analyzing open source projects and also conduct interviews with developers to have better insights about why some developers misidentified the design smell types. Another threat to validity could be related to the severity of the smell occurrences presented in this study. We mitigated these threats by selecting representative instances for each smell type that were discussed and approved by the authors of the smells. We selected examples that are easy to understand, not ambiguous with adequate documentation, and that follow the simple rules related to the definition of the smells. We provided an equal understanding of the smells. We used code examples that reflect commonly used language features and libraries to make it easy to understand. We provided additional support and documentation to avoid additional bias related to the nature and definition of the smell types so that developers can have an equivalent understanding of the different types of smells.

Threats to Internal Validity One of the main threats with a survey is that developers misunderstand the questions and–or possible answers. To minimise this threat, we relied on the literature to extract the possible questions and their answers (for close questions). We also relied on the literature to design the whole study [47]. We provided, in the survey preamble, the definition of all the terms used (*e.g.,* quality attributes, multi-language systems, and design smells). For most of the survey questions, we allowed developers to select a "null" option: "Other (please specify)" if they did not want to answer or wanted to answer differently. Our surveys received a higher response rate than the average response rate in software engineering surveys. This was because of the closed survey, we sent personalized emails, with the developer name and the project name. For the open survey, we also sent personalized messages through LinkedIn. Such specific information increases the chance of contacted developers responding to our email compared to emails with only generic contents. Another reason is that we sent a reminder to developers after two weeks. We have limited a possible bias effect by also showing source code elements without smells to mitigate any bias that could be introduced by developers who could have reported that they perceived the presence of smells even in code not containing any smell. Also some of our results are resulting from the general developers and not the original developer for the smell types *Hard Coding Libraries*, *Excessive Objects*, and *Not Caching Objects*. For these design smell types, our results reflect the perception from the perspective of general developers.

Threats to External Validity These threats concern the generalisation of our results. The results reported in this paper reflect the perception of specific developers and may not generalize to all developers. Also, the impacts and relevance of smells report the perception of the surveyed developers. Such perception depends on the smell instances and could vary from one participant to another. In this study, we had to constrain our analysis to a limited set of smell instances per survey. For the closed survey, we also presented to each participant the smells existing in files he worked on. This may introduce threats because not all the smells were presented to all the developers. However, we are aggregating the results and reporting if the presented smells were correctly identified by the developers. We mitigated the threats to external validity by diversifying the projects analysed. We have also targeted professionals from different countries and backgrounds. We are reporting in this study the perception of software developers in general but also those who contributed to the smelly files. Therefore, we believe that our results could be useful for developers working with JNI projects that contain similar occurrences of design smells. As we target only JNI systems, our results may not generalize to all multi-language systems. However, our results raise awareness about multi-language design smells in general and their impacts on software quality. The results observed in this paper encourage further investigations with other projects and developers to generalize the results of the perceived relevance and severity of multi-language design smells.

Threats to Conclusion Validity They concern the relationship between treatment and outcome. We diversified the keywords and tried to reach developers with different backgrounds and skills. We included general developers but also those who were involved in the development and maintenance of the studied smells. In this paper, we report the results obtained from surveying developers. We were careful to take into account the assumptions of the statistical test used. We used non-parametric tests that do not require any assumption about the data set distribution. To deal with the multiple testing problem, we applied Bonferroni correction and computed its critical value. In this paper, we discuss the results and provide justifications and explanations that are based on our previous research studies and experience with the analysis of multi-language systems [4, 3, 2, 6].

Threats to Reliability Validity They concern the possibility to replicate our study. We mitigate this threat by providing all the information needed to reproduce this study. We discussed in Section 3 the tool used as well as the methodology followed to perform this study. We also explained all the different steps followed to collect and analyze the data[17].

[17] `https://github.com/ResearchML/Replication_EMSE`

7 Related Work

Multi-language Systems Several studies in the literature discussed multi-language systems. One of the very first study, if not the first, was by Linos *et al.* [34]. They presented *PolyCARE*, a tool that facilitates the comprehension and re-engineering of complex MLSs. *PolyCARE* seems to be the first tool with an explicit focus on multi-language systems.

Kullbach *et al.* [29] also studied program comprehension for multi-language systems. They claimed that program understanding for multi-language systems is an essential activity during software maintenance and that it provides a large potential for improving the efficiency of software development and maintenance activities. They presented an approach to support the understandability of multi-language systems.

Linos *et al.* [35] 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. They studied the MLPD concept (Multi-Paradigmatic Program Dependencies), to capture dependencies that arise when entities from one programming language interact with entities developed in another programming language.

Li and Tan [31], on the other hand, highlighted the risks caused by the exception mechanisms in Java, which can lead to failures in JNI implementation functions and affect security. They studied the bugs caused by a lack of management of the exceptions. They argued that such issues negatively impact the security of the system and introduce failures. They proposed a static analysis tool to examine and report potential risks in JNI systems. They focused mainly on JNI but their approach could be adapted to other FFIs, such as Python/C and OCaml/C APIs. They proposed a static-analysis tool to report potential risks in JNI systems. They also proposed a static framework, JET [32], which extends Java's exception checking mechanism to cover native code to prevent failures and ease debugging.

Neitsch *et al.* [46] performed a qualitative study on five multi-language systems. They identified issues related to the deployment of multi-language systems and specified some common build patterns and anti-patterns for multi-language systems that summarise the key problems related to the build of multi-language systems, *i.e., Filename Collision, Installation Required, Unverified Third Party Software, Ignored Error, Incorrect Dependencies, Build-Free Extensibility, Object-Oriented Builds.*

Design Smells 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 reported 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.

Muse *et al.* [43] 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.

Khomh *et al.* [22] investigated the impact of 13 design smells in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino. They reported that classes with design smells are more change- and fault-prone compared to classes without design smells.

Saboury *et al.* [50] 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.

Abbes *et al.* [1] investigated the impact of occurrences of design smells in the developers' understandability of systems while performing comprehension and maintenance tasks. They conducted three experiments to collect data about the performance of developers and study the impact of Blob and Spaghetti Code anti-patterns and their combinations. They concluded that the occurrence of one anti-pattern does not significantly impacts comprehension while the combination of the two design smells negatively impact program comprehension.

Survey Studies Mayer *et al.* [38] performed a survey to investigate multi-language development. Developers reported that multi-language systems introduce benefits and challenges. However, in their study, they focused on the identifiers and cross-language links between languages and did not cover the challenges and developers' practices for multi-language systems. This study also targeted German developers mainly from the same company. However, some of the questions were targeting concrete software project. Thus, respondents in their survey may have reported on the same project.

Yamashita *et al.* [60] conducted a survey aimed at investigating developers knowledge about design smells in mono-language systems and their perceived criticality. They surveyed a total of 85 professional developers and reported that 32% of the participants stated that they did not know about design smells. They reported that the perceived criticality differs from one type of design smell to the other, and that the majority of the participants were moderately concerned by design smells.

Palomba *et al.* [47] conducted a survey aimed at providing evidence on how developers perceive design smells. They surveyed master's students, general developers, but also original developers who contributed to the smelly files. They reported that some design smells instances are generally not perceived by developers as containing a design or implementation problem. Their results also suggest that developers' experience plays an important role in the identification of design smell instances.

Borrelli *et al.* [10] 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 related to the video game. However, they were less concerned by maintainability issues.

Xu *et al.* [58] conducted an open and a closed survey targeting professional developers but also original developers that contributed to the studied systems. They investigated the reasons behind library reuse and library re-implementation. They reported that library reuse results mainly from situations where developers were initially aware and familiar with the library. For the library re-implementation, they reported situations where the used library method is deprecated or a small part of the library, or the library dependencies are complex.

Arcoverde *et al.* [7] investigated how developers react to the presence of design smell occurrences. They reported through their survey that design smells often remain in the source code for a long period of time. They reported that one of the main reasons for postponing the refactoring of design smells is to avoid API modifications.

In contrast to the studies discussed in this section, our study presents the first survey studying the relevance and impacts of multi-language design smells on multi-language software quality. We believe that our study could help developers and researchers improving the quality of multi-language systems and to conduct further research.

8 Conclusion

We presented in this paper the results of our survey of professional developers that aimed to access their perception of the prevalence and severity of multi-language design smells. We also reported about developers' perception of the impact of the design smells on some quality attributes. We selected respondents from different backgrounds to diversify the population under study.

Our results show that in general developers consider the studied multi-language design smells to be reflective of design and implementation problems. We found that the main reasons behind the introduction of multi-language design smells are: refactoring and maintenance, continuous development (*i.e.,* perform regular development tasks), easy way of implementation, lack of knowledge, and specific implementation/design choices. Our results show that multi-language design smells are perceived in general to negatively impact all the studied quality attributes. The design smells perceived as the most harmful are: *Not Handling Exceptions*, *Assuming Safe Return Value*, *Local References Abuse*, *Memory Management Mismatch*, and *Excessive Inter-language Communication.* Our results also show that in general, the developers would consider refactoring the design smells studied in this paper. These findings confirm and complement our earlier empirical evidence [6] and highlight the importance of further research on the impacts of multi-language design smells.

In future work, we plan to (1) study the correlation between bug-proneness and the degree of harmfulness of the design smells, (2) contrast the perception of the prevalence of design smells versus the prevalence in open source projects (3) identify more challenges and issues related to multi-language systems, (4) provide a taxonomy of bugs related to multi-language systems, and (5) study the relationship between multi-language design smells and the taxonomy of bugs.

Acknowledgment

Our sincere gratitude goes to the anonymous respondents who answered the open and closed surveys. We also thank the anonymous reviewers for their comments and suggestions that significantly improved this paper.

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., Openja, M., Khomh, F.: Multi-language design smells: A backstage perspective. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 615–618 (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) **30**(3), 1–56 (2021)
7. Arcoverde, R., Garcia, A., Figueiredo, E.: Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the 4th Workshop on Refactoring Tools, pp. 33–36 (2011)
8. Baltes, S., Diehl, S.: Worse than spam: Issues in sampling software developers. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–6 (2016)
9. Binkley, D.: Source code analysis: A road map. In: Future of Software Engineering, 2007. FOSE '07 (2007)
10. 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)
11. Burow, B.D.: Mixed language programming. In: Computing in High Energy Physics' 95: CHEP'95, pp. 610–614. World Scientific (1996)
12. Emerson, P.: The original borda count and partial voting. Social Choice and Welfare **40**(2), 353–358 (2013)
13. Fink, A.: The survey handbook, vol. 1. Sage (2003)
14. Flores, E., Barrón-Cedeño, A., Rosso, P., Moreno, L.: Towards the detection of cross-language source code reuse. In: Proceedings of the 16th International Conference on Natural Language Processing and Information Systems. Springer-Verlag (2011)

15. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology **11**(2), 5–1 (2012)
16. Goedicke, M., Neumann, G., Zdun, U.: Object system layer. 5th European Conference on Pattern Languages of Programms (EuroPLoP '2000) (2000)
17. 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)
18. Gravetter, F.: Forzano, lab research methods for the behavioral sciences (2012)
19. Harman, M.: Why source code analysis and manipulation will always be important. In: 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, pp. 7–19 (2010)
20. 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)
21. 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)
22. 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)
23. Khomh, F., Guéhéneuc, Y.G.: Do design patterns impact software quality positively? In: Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, pp. 274–278. IEEE (2008)
24. 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)
25. Kienle, H.M., Kraft, J., Müller, H.A.: Software reverse engineering in the domain of complex embedded systems. In: Reverse Engineering-Recent Advances and Applications. InTech (2012)
26. 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)
27. 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)
28. 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)
29. 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)
30. Lee, B., Hirzel, M., Grimm, R., McKinley, K.S.: Debug all your code: Portable mixed-environment debugging. SIGPLAN Not. **44**(10), 207–226 (2009)
31. 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)
32. Li, S., Tan, G.: Jet: Exception checking in the java native interface. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pp. 345–358. ACM (2011)
33. Liang, S.: Java Native Interface: Programmer's Guide and Reference, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
34. 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)
35. 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)
36. Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)

37. 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)
38. Mayer, P., Kirsch, M., Le, M.A.: On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers. Journal of Software Engineering Research and Development **5** (2017)
39. Mayer, P., Schroeder, A.: Cross-language code analysis and refactoring. In: Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on, pp. 94–103. IEEE (2012)
40. Mouna, A., Foutse, K., Guéhéneuc, Y.G.: Anti-patterns for multi-language systems. In: 24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany. ACM (2019)
41. Mouna, A., Manel, G., Foutse, K.: Behind the scenes: Developers' perception of multi-language practices. In: 29th Annual International Conference on Computer Science and Software Engineering (CASCON'2019). ACM (2019)
42. Mouna, A., Manel, G., Foutse, K., Yann-Gaël, G.: Code smells for multi-language systems. In: 24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany. ACM (2019)
43. 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)
44. 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)
45. 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)
46. 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)
47. 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)
48. 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)
49. 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)
50. 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)
51. Sheskin, D.J.: Handbook of parametric and nonparametric statistical procedures. crc Press (2020)
52. 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)
53. Synytskyy, N., Cordy, J.R., Dean, T.R.: Robust multilingual parsing using island grammars. In: Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '03, pp. 266–278. IBM Press (2003)
54. 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)
55. 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)

56. 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)
57. 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. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 403–414. IEEE (2015)
58. Xu, B., An, L., Thung, F., Khomh, F., Lo, D.: Why reinventing the wheels? an empirical study on library reuse and re-implementation. Empirical Software Engineering **25**(1), 755–789 (2020)
59. 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)
60. 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)
61. Yin, R.K.: Applications of Case Study Research Second Edition (Applied Social Research Methods Series Volume 34). {Sage Publications, Inc} (2002)
62. Zhang, C., Budgen, D.: What do we know about the effectiveness of software design patterns? IEEE Transactions on Software Engineering **38**(5), 1213–1231 (2012)

9 Appendix

We provide in this section the details about the detection results and the smell occurrences detected in our subject systems presented in Table 1. We also highlight the total number of responses sent and received from running the survey.

Table 11 provides the smells occurrences detected when running `MLSInspect` [6] on the 270 snapshots of our subject systems described in section 3. The columns reported in Table 11 reflects the subject systems analysed. The first row (#Snap) reports the number of snapshots in each system, while the rest of the rows describe the occurrences of smells detected in each system.

Table 11: Overview of the Detected Smell Occurrences in the Studied Systems

Projects	**Frostwire**	**OpenDDS**	**conscrypt**	**javacpp**	**jna**	**pljava**	**realm-java**	**rocksdb**
#Snap	18	58	32	30	32	35	29	36
NHE	32	262	42	0	6	35	134	63
UP	2206	1935	266	9	193	540	493	786
NSL	103	218	53	30	5	2	28	68
ASRV	16	219	42	0	1	0	45	63
LRA	0	43	22	0	6	7	0	19
NRP	87	44	42	0	5	2	0	27
MM	0	0	21	0	5	28	22	28
HCL	0	0	0	7	0	0	0	21
EO	0	0	0	0	0	0	0	0
NCO	0	0	0	0	0	0	0	4
TMC	37	246	62	67	12	135	203	480
TMS	0	577	0	349	16	530	114	798
EXC	32	0	44	257	10	61	174	287
UMD	55	264	69	124	1	574	54	138
UMI	15	107	0	0	1	37	45	0

Table 12 summarizes the total number of surveys that we sent, the number of answers received, and the number of answers we kept for the analysis of this paper. Note that we did not keep answers that contain information related only to the background section as those participants did not fill any of the sections related to the purpose of this study. Another important point is that in this study, we initially aimed to capture results for nine systems as stated in our initial protocol [5]. However, the results we received are only for eight systems as shown in Table 12.

Table 12: Survey Responses Overview

Survey	Total Sent	Total Answers Received	Total Answers Kept
Open	500	216	132
Frostwire	10	4	3
OpenDDS	15	5	5
Conscrypt	34	8	7
JavaCpp	9	4	4
JNA	5	2	1
Pljava	11	6	4
Realm	30	9	6
Rocksdb	123	12	9
vlc-android	19	0	0
java-smt	7	0	0