Code Smells for Multi-language Systems

Mouna Abidi
Polytechnique Montreal
mouna.abidi@polymtl.ca

Manel Grichi
Polytechnique Montreal
manel.grichi@polymtl.ca

Foutse Khomh
Polytechnique Montreal
foutse.khomh@polymtl.ca

Yann-Gaël Guéhéneuc
Concordia University
yann-gael.gueheneuc@concordia.ca

ABSTRACT

Software quality becomes a necessity and no longer an advantage. In fact, with the advancement of technologies, companies must provide software with good quality. Many studies introduce the use of design patterns as improving software quality and discuss the presence of occurrences of design defects as decreasing software quality. Code smells include low-level problems in source code, poor coding decisions that are symptoms of the presence of anti-patterns in the code. Most of the studies present in the literature discuss the occurrences of design defects for mono-language systems. However, nowadays most of the systems are developed using a combination of several programming languages, in order to use particular features of each of them. As the number of languages increases, so does the number of design defects. They generally do not prevent the program from functioning correctly, but they indicate a higher risk of future bugs and makes the code less readable and harder to maintain. We analysed open-source systems, developers' documentation, bug reports, and programming language specifications and extracted bad practices related to multi-language systems. We encoded these practices in the form of code smells. We report in this paper 12 code smells.

KEYWORDS

Code smells, multi-language systems, code analysis, software quality

ACM Reference Format:
Mouna Abidi, Manel Grichi, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Code Smells for Multi-language Systems. In *24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361149.3361161

1 INTRODUCTION

Nowadays, most of the systems are written using a combination of several programming languages and technologies. The core of an application might be written in Java, while it has some routines written in C, with user interface written in PHP, JavaScript, and HTML [1]. Most of the systems with which we interact daily are built using a combination of programming languages, such as Facebook, Youtube, etc [2]. Developers can reuse existing modules without re-implementing the source code from scratch [3]. They often choose the programming language suitable for their needs, instead of having all the tasks written in a single language [4–6]. Consequently, software systems became more complex, and their maintenance becomes more challenging [7].

software quality presents one of the most important concerns during the software development, providing software with high quality could reduce maintenance and testing cost [8, 9]. Software quality has been widely studied in the literature and has been often directly related to the presence of design patterns, anti-patterns and code smells. Design patterns are defined in the GOF as reusable good solution to recurring design problems [10]. Design defects are known as the opposite of design patterns [11]. They include anti-patterns, which are higher-level design defects, and code smells, which are lower-level defects [12]. Code smells include low-level problems in source code, poor coding decisions that are symptoms of the presence of anti-patterns in the code. As code smells are error-prone and change-prone, it is important to detect and correct them as soon as possible.

Several studies in the literature studied occurrences of code smells and their impact on software quality with studies mostly focusing on a single programming language [13–16]. Few studies investigated such good or bad practices for multi-language systems [3, 17]. Some studies also focused on the design patterns related to such systems [18–21]. With the aim of improving the software quality of multi-language systems, we mined open-source systems, developers' documentation, bug reports, and programming language specifications. We observed and cataloged bad practices related to the development and maintenance of multi-language systems. The systems we analysed contain mainly Java/C(++) but they are also developed using other programming languages. We observed multi-language practices at the implementation level, we cataloged and documented these practices in the form of code smells. These code smells could apply to cloud applications, microservices or to their implementation, and to any other pieces of code written with more than one programming language. These multi-language files do not have to compile in a single binary file, they just have to interact with each other.

The remainder of this paper is organised as follows. Section 2 discusses the background of multi-language systems, code smells, and related works. Section 3 describes our methodology for gathering the code smells. Section 4 reports multi-language systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '19, July 3–7, 2019, Irsee, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6206-1/19/07…$15.00
https://doi.org/10.1145/3361149.3361161

code smells. Section 5 summarises threats to the validity. Section 6 concludes the paper and discusses future works.

2 BACKGROUND AND RELATED WORK

We now describe a background about multi-language systems and code smells. We later discuss some related works.

Multi-language Systems: Nowadays software applications are moving from the usage of a single programming language towards the combination of several programming languages. Developers often choose the "best" programming language suitable for their needs instead of implementing all the code with a single programming language [2]. The benefits of reusing existing code is an increasingly powerful reason behind the increase of multi-language systems usage.

Code Smells: Design defects are known as the opposite of design patterns, they are defined as a bad solution to a recurring problem. "Bad", in this context, means that, for instance, the chosen solution can be ineffective, make the code unclear and difficult to maintain, yet maintenance is costly. They include anti-patterns, which are higher-level design defects, and code smells, which are lower-level defects [22, 23]. Code smells include low-level problems in source code, poor coding decisions that are symptoms of the presence of anti-patterns in the code.

Encoding and Cataloguing: Several templates are used in the literature to define patterns and defects. We adapted the template provided by Brown [22] to the specificity of our work as follows:

- Code Smell: It provides the name of the code smell.
- Context: The context in which the code smell could occur.
- Problem: It describes the problem that may lead to the wrong solution.
- Bad Solution: The bad Solution is the solution attempting to solve the problem, but that presents poor decision coding.
- Consequences of the code smell: They describe the impact of using the "bad" solution to solve the problem.
- Refactoring: This solution illustrates the steps to apply to remove the code smell or apply the good solution. This also provides an example off applying the refactored solution.
- Benefits of the Refactoring: These benefits describe the positive impact of applying the refactoring to remove the occurrences of the code smell.
- Examples: These examples describe concrete examples of "bad" coding decision. It presents the code smell in context. These examples are mainly extracted from concrete examples of developer's documentation discussing the bad practice and impact of applying the bad solution.

Related Work: Several studies in the literature investigated the quality of multi-language systems.

Kondoh et al. [24] presented four kinds of common JNI mistakes frequently made by developers. They proposed a static-analysis tool to retrieve JNI mistakes pertaining to error checking, virtual machine resources, invalid local references, and JNI methods in critical sections of the code.

Osmani et al. [25] introduced the Lazy Initialisation pattern which describes guidelines to execute Ajax requests in JavaScript. The Ajax request includes a URL and some data, possibly in JSON or XML, to communicate with a server, likely implemented in C/C++.

Li and Tan [26] studied the risks existing in JNI systems and proposed a pattern of mishandling the exceptions. 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 discussed their pattern in the case of JNI system but argued that it can be applied with other kinds of FFIs.

Tan et al. [3] defined JNI bug patterns extracted from the JDK. They examined a range of bug patterns in the native code and identified six bugs related to the use of JNI methods in the JDK. The reported causes may lead to a JVM crash or can introduce other vulnerabilities. They argued that bugs are likely to occur due to the difference and incompatibilities between programming languages. They also discussed the assumptions made by the Java code regarding the C(++) code, including, the native method *java.util.zip.Deflater.deflatesByte()* which assumes that its Java callers check bounds, which could lead to buffer overflows.

Ayers et al. [27] collected bugs in multi-language systems. They proposed TraceBack a tool that collects bugs in multi-languages systems by storing data through runtime instrumentation of control-flow blocks. The approach is based on intermediate language representation to provide a unified trace of components' execution.

Mayer and Schroeder [28] investigated the dependencies existing in multi-language systems code. They proposed a technique to automatically retrieve dependencies among multi-languages modules, warn of potential missing dependencies, and also propagate renaming among multi-language source code.

3 STUDY DESIGN

In this section, we present the steps followed to collect the code smells.

We provided in our paper focusing on anti-patterns of multi-language systems a deep presentation of all the steps used for the data collection, analysis, and documentation of the anti-patterns and code smells of multi-language systems [29]. The figure 1 presents an overview of the methodology to collect the anti-patterns and code smells.

We started by setting the objective of this study. Our objective in this paper is to collect and document code smells for multi-language systems. For that, we mined all possible sources of information that we could access, including developers' blog and bug reports. We previously performed a systematic literature review on multi-language systems and found that the most used combination of languages is Java/C(++). Thus, we decided to start our research with this set of programming languages and then include more programming languages. We collected common practices and guidelines discussed in the Java Native Interface specification [17] and developers' documentation. We also studied common pitfalls made by developers and reported in developers' blogs and bug reports[2] [1]. From all these sources of information, we build a list of possible practices and documented them in terms of definition, context, and examples.

We then considered the case of multi-language systems in general, systems with more than one programming language. We

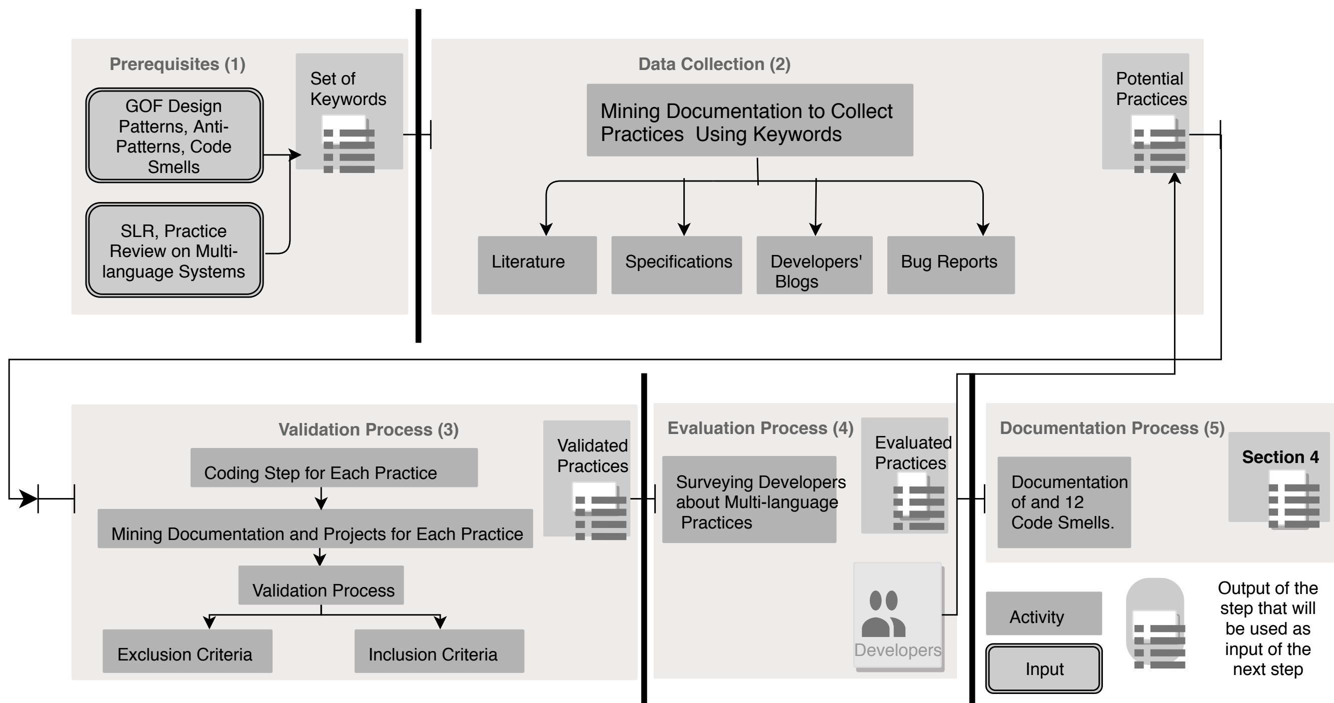

Figure 1: Overview of the Methodology Used to Collect and Document the Anti-patterns and Code smells for Multi-language systems.

searched for any possible issues reported by developers and related to multi-language systems. We defined a set of common keywords related to issues in multi-language systems and queried these sources of bugs. We used the following list of keywords *JNI issue, foreign library, Python/C issue, API, polyglot, incompatibility, compilation errors, programming languages issues, memory issues, security issues, performance issues, foreign function interface, etc.* We deeply searched for these keywords in well-known websites such us *Stack Overflow*, *GitHub issues*, *Bugzilla*, *IBM Developers*[1], and *developer.android*[2]. We studied the issues to understand whether the reported issue is related to the combination of more than one programming language or whether it is simply focusing only in a single language. As examples of code smells extracted from *Bugzilla*, was when searching for *JNI issue*. We had 23 results, among them we considered only two possible bad practices. The first bad practice was related to the loading of the native library[3] and the second related to the management of exceptions[4]. Using *Stack Overflow*, we had 500 results for each of these keywords *JNI issue* and *Python/C issue*, we then searched manually only for issues that have been already discussed in the developers' documentation[2] [1].

We relied on inclusion and exclusion criteria to validate our catalog of code smells. As inclusion criteria, we considered a practice that occurred in at least three different contexts and–or systems. For that, we verified if the good or bad practices discussed in the literature were also used in at least three classes, source code files, or systems. We also considered the case of good practice discussed in the literature and other developers' but was not followed in some open source systems (e.g. The code smell *Not Checking Exceptions* was discussed in several sources of information including developers' documentation[1] and in some articles but was not followed in most of the systems that we analysed). As exclusion criteria, we excluded practices for which, we were not able to find occurrences at least three sources of information, including open source systems. We excluded practices that seem more likely to be a simple developers' habits than potential code smells.

We also used data already extracted in one of our prior studies focusing on JNI usage. The data consists of 100 multi-language open source projects. These projects were mainly developed using the Java Native Interface (JNI) but also include other sets of programming languages. We relied in OpenHub to have the list of all the programming languages used in the project (e.g. *OpenCv* is mainly written in C(++) but contains 25.239 Python lines of code, 24.427 Java lines of code, and other languages). We mainly focused on these systems to perform this study: *libgdx, Openj9, Rocksdb, Google toolkit, JMonkeyEng, PortAudio Java Bindings, OpenVRML, jpostal, Jna, JavaSMT, ZMQ, Telegram, reactNative, OpenCV, JatoVM, Tenserflow, Frostwire, SQLlite, Godot, python-telegram-bot.* We provide in Section 4 the sources and-or name of the projects from which we extracted the code smells. We also performed a survey with professional developers to validate some of the practices that we extracted and also to ask about other practices used. We do

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

not discuss in this paper the results of this survey, as it is used for another study [30].

We reported all the observed practices and had some discussion to validate whether the selected practices are valuable and should be documented in form of code smells, or if they present only some practices or developers' habits. We discussed each case until a consensus was reached.

4 CODE SMELLS FOR MULTI-LANGUAGE SYSTEMS

In this section, we introduce the good and bad practices in the form of code smells.

Passing Excessive Objects.

- **Context:** We have some attributes from classes and objects in the host language that we must access and use in the foreign code.
- **Problem: Developers do not have enough knowledge about the performance cost when integrating several programming languages.** They usually take design and coding choices considering a single paradigm and do not consider that combining distinct paradigms may change those decisions.
- **Bad Solution:** We usually have to decide whether we pass an object that has multiple fields or the fields individually. The bad solution would be to always favor passing a whole object instead of passing parameters; i.e., each time we pass the whole object instead of passing the parameters of interest. If we consider passing the whole object in the context of object-oriented principle, this provides better encapsulation. However, in the case of multi-language systems, it is better to consider the performance cost between the two solutions when combining different paradigms and languages.
- **Consequences of the Code Smell:** Passing an object from one language to another may require an important effort of performance and implicate intermediate methods to access the native code as not all the language have or treat similarly the types. In some cases, the native code uses several foreign calls to get the value of each individual field. Such additional calls add extra costs. Calls from native code to host language code is more expensive than a normal method call and may negatively impact the performance. Other consequences are that the methods implicated by this code smell will not have many parameters and will favor the encapsulation.
- **Refactoring:** To remove this code smell, **a good solution would be when few parameters are needed to be accessed, favor passing them separately instead of passing the whole object.** Depending on the languages, it may require additional effort to access the fields if they are not passed as parameters.
- **Benefits of the Refactoring:** This will improve the performance in the case where passing a whole object is a consuming task. It also improves the readability by having the parameters of interest instead of whole objects. Another benefit is to avoid calling heavy methods to extract the parameters from the object, especially when the programming languages differ in term of types and paradigms.
- **Examples:** An example of occurrences of this code smell has been discussed in *IBM website*[1]. In the case of JNI, when we pass objects, it results in many calls to get the value for each of the individual fields. This kind of calls add an extra cost as the interactions between the native code and the Java code is generally more expensive than a method call. It may negatively impact performance. Figure 3 presents an example of occurrences of this code smell. While figure 4 presents a possible refactoring to remove this code smell. Depending on the programming language this code smell may also occur in Python/C and other sets and pairs of languages.

Unnecessary Parameters.

- **Context:** When adding new features or modifying an existing project, it may happen that we are not sure which parameters to keep and which one to remove. This can also happen when passing parameters to and from one language to another which were never been used in the other language.
- **Problem:** Several teams and developers are involved in the same projects. **These projects are then maintained by other developers that do not have enough knowledge about the architecture of the project**.
- **Bad Solution:** A bad solution would be, when applying a change to always keep the parameters already existing as they may be used in the other language while they are no longer used. This can also appear when we pass all the parameters that we believe can be used to complete the task while concretely not all of them are used.
- **Consequences of the Code Smell:** Having unused parameters from one language to another may add complexity to the code especially in the maintenance activities. Developers may not be sure which parameters should be used and which not as they are related to another language. Multi-language systems are by nature more difficult to understand, adding unnecessary parameters or applying a change and not removing the corresponding parameters will introduce more complexity to the system. Some developers may go through this solution as once all the parameters are defined and passed from one language to the other, it is easier to use them or apply changes that involve these parameters.
- **Refactoring:** To remove this code smell, **Keep only the parameters that are used to avoid introducing unnecessary complexity and improve the readability.**

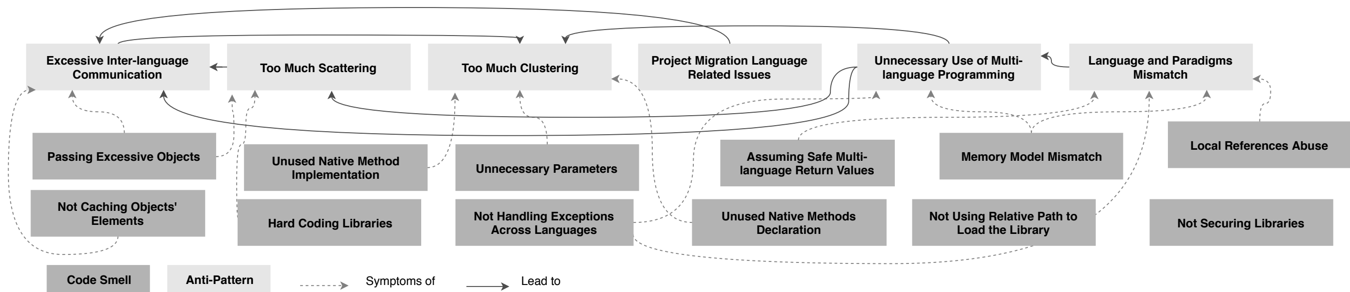

Figure 2: Pattern Overview Diagram

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

Figure 3: Code Smell - Passing Excessive Objects

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

Figure 4: Refactoring - Passing Excessive Objects

- **Benefits of the Refactoring:** Improve the understandability and maintainability as the method will contain only the parameters used. This also avoids dead code and Keep only the parameters needed.
- **Examples:** Figure 5 presents an example of occurrences of this code smell. The parameter *acceleration* is defined in the native method signature. However, it is not used by the native code. The solution would be to remove the unused parameters.

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

Figure 5: Code Smell - Unnecessary Parameters

Unused Native Methods Declaration.

- **Context:** When we have some methods declaration in the host language that has never been implemented in the foreign language.
- **Problem: Requirement or functionalities changes may lead to unused code.** Usually, different teams may be involved separately to contribute in each programming language. These teams do not have a global view of the whole system, which methods are used and which are not.
- **Bad Solution:** A bad solution would be when applying a change to always keep the native methods declared without additional checking as they may be used in the other language while they are no longer used.
- **Consequences of the Code Smell:** If a future modification involves implementing these methods, it will be easier as they are already declared. However, this code smell can result in unused and unnecessary code. It may add some complexity to the code and introduce more difficulty when reading and maintaining the code. Depending on the languages, this kind of methods may not crash the system or display an error, as these methods are never called or used. However, for a maintainer, it would require additional effort to investigate which methods are really used in the multi-language systems and which are not.
- **Refactoring:** To remove this code smell, **keep only the methods that are used in the multi-language systems' interaction.** An unused code may negatively impact the quality of a system, the impact may be important when we are dealing with multi-language systems. Depending on the size of the system, it may be difficult for a maintainer to identify the methods used. To retrace or fix a bug this may require more effort.
- **Benefits of the Refactoring:** Improve the understandability and maintainability as the code will contain only the methods that are used. This also avoids dead code by providing clean code and Keeping only the methods used. Another benefit is that it would be easier for a maintainer or new developer to locate the code used.

- **Examples:** An example of this code smell has been perceived when we analysed JNI systems and collected the number of method implementation and the number of a method declaration. In most of the system, the number was the same between both of these metrics. However, we found examples where some native methods have been declared but have never been used.

Unused Native Method Implementation.

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

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

- **Bad Solution:** Always keep the native methods implementations without additional checking as they may be used in the other language. Avoid breakages related removing code that is still called or used somewhere on the project.

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

- **Refactoring:** To remove this code smell, **remove all unnecessary and unused code to reduce the complexity and keep in each class only the methods that are really used.** To prevent occurrences of this code smell, it is also important to always remove all the code related to the multi-language programming if it is no longer used. As these systems usually involve different developers or teams working separately and it may be more difficult for them to know if the code is used somewhere in the project or not.

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

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

Not Handling Exceptions Across Languages.

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

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

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

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

- **Refactoring:** To remove this code smell, **always check whether an exception has been thrown after invoking any foreign methods that may throw an exception.** Multi-language systems introduce more complexity than mono-language systems and need more effort to fix bug and issues, it is important to consider checking and handling exceptions to prevent issues related to no checking exception. In the case of multi-language systems, it is much easier to prevent crashes by implementing the exception than to debug after the crash occurred. Upon handling the exception, we should also clear it depending on the language. For JNI, we should use the *ExceptionClear* function to inform the

Java VM that the exception is handled and JNI can resume serving requests to Java space. If the host language provides the handling and management of exceptions, it possible to simply check if an exception has occurred in the foreign code and if so return immediately to the host code so that the exception is thrown. It will then be either handled or displayed using the exception-handling process provided by the host language.

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

- **Examples:** Examples of occurrences of this code smell have been discussed in developers' documentation as a wise practice[1] [5]. Most of the systems that we analysed were not always implementing a proper way to handle the exception as shown in figure 6, this code may cause a crash if charField field no longer exists. For the JNI case, one good example was *Libgdx*, where they catch Java exceptions in native code using the JNI API call ExceptionOccurred. Figure 7 presents a refactoring example extracted from *IBM Developer Site*[1]. Occurrences of this code smell will not block the execution of the native code. However, any calls to JNI API will silently fail. As the actual exception does not leave any traces behind, it is hard to debug.

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

Figure 6: Code Smell - Not Handling Exceptions Across Languages

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

Figure 7: Refactoring - Not Handling Exceptions Across Languages

Assuming Safe Multi-language Return Values.

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

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

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

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

- **Refactoring:** To remove this code smell, **a good solution would be to never assume that it is safe to use a value returned by a language API call, which must always be checked to make sure that the call was successfully executed and the proper usable value is returned to the native function**. Multi-language methods usually have a return value that indicates whether the call succeeded or failed. A common bad practice, similar to not checking for exceptions, is to assume that the return values are safe. API functions rely on their return values instead to indicate any errors during the execution of the API call.

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

- **Examples:** Examples of occurrences of this code smell have been observed in most of the open source systems that we analysed. This was also reported in several developers' documentation and bug reports[6]. Depending on the languages involved in the multi-language systems, it is mostly recommended to always check the return values from one language to another. As in most of the cases, we use another language

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

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

to perform a calculation or specific features that will be then used by the main language. It is recommended to always check the value before returning it to the host language. As illustrated in figure 8 extracted from *Libgdx*, if the class *NIOAccess* or one of its methods is not found, the native code will cause a crash. As we are not applying any check to handle the problems related to the return values. A good solution to remove this code smell is to add a check that handles the situations in which problems may occur with the return values. Figure 9 is a good example to illustrate a possible refactored solution.

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

Figure 8: Code Smell - Assuming Safe Multi-language Return Values

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

Figure 9: Refactoring - Assuming Safe Multi-language Return Values

Not Caching Objects' Elements.

- **Context:** When implementing a multi-language system, we need to pass objects and variables from one language to the other. In this case, we want to access an object's field and methods from the foreign code.
- **Problem: Developers do not have enough knowledge of how the fields and methods are retrieved when passing from one language to another.** They may consider using the most simple way to access foreign code and do not consider the performance cost.
- **Bad Solution:** Depending on the language, use the available methods to access the objects fields and methods. Each time we need one of the object's field, call the methods to retrieve the field as if it is the first time we access the field. As example to access Java objects' fields and their methods, the native code perform calls to FindClass(), GetFieldID(), GetMethodId(), and GetStaticMethodID().
- **Consequences of the Code Smell:** Depending on the language, it may require an important effort to use available methods to access the object fields and methods. Although these methods may be used frequently in multi-language applications, they may be heavy function calls by their nature. These functions traverse the entire inheritance chain for the class to identify the ID to return. The IDs returned for a class using GetFieldID(), GetMethodID(), and GetStaticMethodID(), do not change during the lifetime of the JVM process. However, this may be expensive in term of performance. For that, we recommend to look them and reuse them once needed.
- **Refactoring:** Neither the Class object, the Class inheritance, nor the fieldID can be changed during the execution of the system. These values are cached in the native layer for subsequent accesses. The return type of the FindClass function is a local reference, so to cache its values, developers must create a global reference first through the NewGlobalRef function when it is needed. The return value of GetFieldID is jfieldID, which is an integer that can be cached as it is. To remove this code smell, **developers should focus on caching both the field and method IDs that are accessed multiple times during the execution of the application,** this practice makes an improvement in the execution time.
- **Benefits of the Refactoring:** The IDs are often pointers to internal runtime data structures. Looking them up may require several string comparisons. Once we have them the call to get the field or the method does not take an important time. This also improves performance by avoiding several lockups and avoid calling heavy functions.
- **Examples:** Examples of occurrences of this code smell have been observed in JNI systems. Some developers' documentation also reported this common bad practice as negatively impacting the performance as shown in figure 10[1]. In the case of JNI, a correct way to initialise the IDs is to Create a method in the C(++) code that performs the ID lookups. The code will be executed once when the class is initialised. If the class is ever unloaded and then reloaded, it will be executed again. If commonly used classes, fields Ids, and methods Ids are not properly cached, we lose the benefit of using the C(++). This code smell negatively impacts the performance. As presented in the example[1], using caching field IDs will take 3,572 ms to run 10,000,000 times 10. However, without using the cache as illustrated in 11, it takes 86,217 ms. Using this code smell the task takes 24 times longer than without the occurrences of this code smell.

Not Securing Libraries.

- **Context:** We want to access foreign libraries or an API available in another language. We aim to integrate an external

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

Figure 10: Code Smell - Not Caching Objects' Elements

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

Figure 11: Refactoring - Not Caching Objects' Elements

library with the main application developed in a different language.

- **Problem: Developers are not always aware of the consequences of insecure code** or do not provide enough intention.
- **Bad Solution:** When developing multi-language systems, we always need to access some API or libraries implemented in another language. We load the native library or API directly in the code without any security checking or restriction.
- **Consequences of the Code Smell:** As consequences of the occurrence of this code smell, several problems may occur due to the leak of security. An unauthorised code may access and load the libraries. Malicious code may use this vulnerable code to access the system. Depending on the domain of application in which the multi-language systems has been involved, this may have an important impact. As for mobile application or embedded systems, a fault in the security may have an impact at the human level.
- **Refactoring:** To remove this code smell, **always ensure that the libraries cannot be loaded without permissions.** It is important to ensure that the loading of external libraries is written in a secured block of code to guarantee access only to those who are allowed to. Depending on the language some predefined classes may ensure security and prevent undesirable access to the system. As for the Java language, it is recommended to always load libraries in static blocks, wrapped in a call to *AccessController.doPrivileged* or use the *securityManager*.
- **Benefits of the Refactoring:** One of the main benefits is to ensure that the libraries cannot be loaded without permissions. This also avoids malicious attacks and secure the load of the library and the project.
- **Examples:** The occurrences of this code smell have been observed on most of the analysed systems. In the JNI case, we found the usage of the secure library only with the *JDK* and *Openj9*. In these systems, the loading library is always performed in a static block and using the *AccessController*. *AccessController* presents a safe way to load a library because it ensures that the library cannot be loaded without permissions, as shown in figure 12. Depending on the languages, we recommend to always secure the loading library by using available methods for the specific language.

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

Figure 12: Securing Library Loading

Hard Coding Libraries.

- **Context:** We are loading different libraries for different OS, the same code can not run in all the platforms. we need to customise the loading according to the OS. For that, we hard-code the loading according to the OS.
- **Problem:** Project was designed as a prototype and do not consider future extensions and adaption to new platforms.
- **Bad Solution:** Depending on the used language, some of them are expected to run on all platforms, but in other languages, there must be different native code libraries for different platforms, which must be loaded according to the target OS. To ensure loading the libraries according to the OS the loading libraries is hard-coded in the code.
- **Consequences of the Code Smell:** When the libraries are hard-coded, it is difficult for a maintainer to know which library is loaded in which time. Even to handle bugs and errors this would require more time to locate the errors. As consequences of this code smell, developers may require additional effort to distinguish between the different libraries. This also may impact the understandability and readability of the system.
- **Refactoring:** To remove this code smell, **a clean way to load the library would be to handle all targeted OS on which the library is available.** This ensures better code readability, letting the code Reader directly knows what libraries are being loaded. Also, loading in a way to take care

of the OS makes sure that all cases are properly covered and if a code is running on a new OS, errors are easy to locate.

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

- **Examples:** Examples of occurrences of this code smell as well as the good solution has been observed respectively in `JavaSmt` and `Frostwire`. In `JavaSmt`, most of the loading libraries were hard-coded in a way that it was difficult for us to know which library is related to which os. As shown in figure 13. Some of the comments were explaining the OS related to the library. However, it is better if the way to load the library can be self-efficient and reflect which library is loaded. It is important when loading libraries to take care of the OS as shown in figure 14. This ensures that all platforms are covered and those missing libraries can be easily identified.

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

Figure 13: Code Smells - Hard Coding Libraries

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

Figure 14: Refactoring - Hard Coding Libraries

Not Using Relative Path to Load the Library.

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

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

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

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

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

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

- **Examples:** We perceived examples of occurrences of this code smell when analysing JNI systems. Only a few of the systems that we analysed followed the practice of using a relative path. The systems `Conscrypt` and `JatoVm` are mainly relying on the relative path to load the library, while most of the systems that we analysed only specify the name of the library. Figure 15 presents an example of refactoring to remove this code smell extracted from `JatoVm`.

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

Figure 15: Refactoring - Not Using Relative Path to Load the Library

Memory Management Mismatch.

- **Context:** We are implementing a multi-language system in which we are passing reference types from one language to another. Depending on the languages, these types may be considered as pointers when used in other languages.
- **Problem: The management of the types and memory is not the same from one language to another.** In some languages as the C(++), the management of the memory is not done automatically. Depending on the language, it may be the developers' responsibility to care about the management of the memory. As in the case of JNI, if we are using a *String* we should be the one taking care of releasing it after its usage. However, developers do not have enough knowledge of the characteristics of programming languages involved. They are usually dealing with programming languages that automatically handle the management of the memory.
- **Bad Solution:** The bad solution would be to to not consider the differences and possible incompatibilities between different programming languages when managing the memory. Rely on the management of memory provided by a single programming language without additional checks to confirm that they memory is correctly managed.
- **Consequences of the Code Smell:** The management of the memory may not be the same from one language to another. Memory leaks can occur if the developers forget to take care of releasing such reference types.
- **Refactoring:** To remove this code smell, a good solution would be to **always take care of the management of such references types.** It is better to assume that in foreign communication, the management of the memory is not always done automatically, and may be considered by the developers. Especially the allocation and release of memory that needs to be explicitly done by the developers. It becomes their responsibility when dealing with more than one programming language.
- **Benefits of the Refactoring:** One of the main benefits is to avoid problems due to a leak of the memory. This also avoids performance issues and free the memory allocated to objects that are no longer used.
- **Examples:** Examples of occurrences of this code smell have been observed in few JNI systems and developers' documentation, where problems related to the leak of memory occurred due to not releasing the memory. Developers' should take care of such memory management in multi-language systems. For the JNI case, Java strings are handled by the JNI as reference types. Those reference types are not null-terminated C char arrays (C strings). When the Java string is converted to a C string, it simply becomes a pointer to a null-terminated character array. It is the developers' responsibility to explicitly release the arrays using the ReleaseString or ReleaseStringUTF functions. Figure 16 presents an example of the refactored solution to remove this code smell. As the example of occurrences of this code smell is the nonrelease of the memory using `ReleaseString` or `ReleaseStringUTF`.

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

Figure 16: Refactoring - Memory Management Mismatch

Local References Abuse.

- **Context:** Depending on the programming language, the management of the memory is not the same. For this code smell, we are considering the references. For the Java code, JVM keeps an eye on the available references to the allocated memory regions. When JVM detects that an allocated memory region can no longer be reached by the application code. It releases the memory automatically through garbage collection leaving developer free from memory management. But JVM garbage collectors boundaries are limited to the Java space only.
- **Problem:** The lifespan of a local reference is limited to the native method itself. **Depending on the language, garbage collectors boundaries are limited to the specific space only,** so the garbage collector cannot free the memory that the application allocates in the native space. The management of the memory may differ from one language to another. Thus, developers should always consider taking care of memory when using local and global references. For the JNI case, memory models and their management defer between Java and C. It is the developers' responsibility to manage the application's memory in native space properly.
- **Bad Solution:** The bad solution would be to use local and global references without considering the management of the memory.
- **Consequences of the Code Smell:** If we do not consider the management of memory and the criteria when using references from one language to another, this can cause memory leaks.
- **Refactoring:** To remove this code smell, **always take care of releasing the memory once using global or local references and never assume that their release will be done automatically.** For the JNI case, it creates references for all object arguments passed into native methods, as well as all objects returned from JNI functions. These references will keep Java objects from being garbage collected. To make sure that Java objects can eventually be freed, the JNI by default creates local references. Local references become invalid when the execution returns from the native method in which the local reference is created.

- **Benefits of the Refactoring:** One of the main benefits is to ensure releasing the memory once using global or local references. This avoids memory allocation bugs and makes sure to free the memory for reuse when no longer needed.
- **Examples:** Occurrences of this code smell have been observed in JNI systems and the good practice of releasing the memory has been discussed in several developers' documentation as well as the JNI specification. Each time we return an object by a JNI function, local references are created. For example, as shown in figure 17 calling GetObjectArrayElement() will return a local reference to each object in the array. It is important to delete each reference when it is no longer required. A native method must not store away a local reference and expect to reuse it in subsequent invocations. so whenever a state is to be maintained during JNI calls, global references is a must. However, JNI global references are prone to memory leaks, as they are not automatically garbage collected, and the programmer must explicitly free them but they are necessary. Depending on the programming language, to reuse a reference, the developer must explicitly create a global reference based on the local reference using the NewGlobalRef JNI API call. The global reference can be released when it is no longer needed using the *DeleteGlobalRef* function. An example of refactoring is: env->DeleteLocalRef(globalObject).

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

Figure 17: Code smell - Local References Abuse

5 THREATS TO VALIDITY

We now discuss threats to the validity of our methodology to collect the code smells.

Threats to internal validity: We accept this threat as we did not provide an exhaustively list of all existing code smells related to multi-language systems. However, we mined open-source repositories *GitHub* and *OpenHub* to identify the list of multi-language systems. We also queried and analysed bug reports and developers' blogs, such as *StackOverflow*, *IBM Developers*, *developer.android* to extract practices.

Threats to external validity: We tried to minimise these threats by reporting code smells observed at least more than three times in various systems or documentation. However, depending on the sets of programming languages used, some of these code smells may not be existent or may have different consequences.

Threats to reliability validity: We mitigate the threats to reliability by providing online access to all the data and scripts that we used to conduct this study. All the information are available in the companion website[7] [8].

6 CONCLUSIONS AND FUTURE WORK

The development of multi-language systems has become very prevalent nowadays, it offers many opportunities, developers can reuse existing code and take advantage of existing libraries and modules written in several programming languages [2]. Such multi-language systems are difficult to analyse and maintain. They introduce several challenges that developers are continuously facing when dealing with such systems.

Software quality is achieved partly by following good practices and avoiding bad ones. While several studies discussed the benefits and challenges of multi-language systems, only a few reported good and bad practices that developers should adopt when dealing with such systems [18–21]. Good and bad practices related to the development, maintenance, and evolution of multi-language systems are scattered across different resources, including few academic papers, developers' documentation, programming-language specifications, etc.

Therefore, through this paper, we collected, cataloged, and documented the practices observed in the form of code smells. We report in this paper 12 code smells that we borrowed and observed from these resources. These practices should help not only researchers but also developers involved in the development of multi-language systems.

In future work, we will (1) survey developers about these code smells, (2) provide an exhaustive catalog of multi-language code smells, (3) investigate the impact of these code smells on some quality attributes.

ACKNOWLEDGMENTS

We thank our shepherd Uwe Zdun for his valuable suggestions that significantly improved this paper. We also would like to thank the review group on EuroPLoP conference '19. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 3, p. 12, 2009.
[2] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 563–573.
[3] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 365–377.
[4] F. Tomassetti and M. Torchiano, "An empirical assessment of polyglot-ism in github," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 17:1–17:4.
[5] R.-H. Pfeiffer and A. Wąsowski, "Texmo: A multi-language development environment," in *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ser. ECMFA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 178–193.
[6] Z. Mushtaq and G. Rasool, "Multilingual source code analysis: State of the art and challenges," in *2015 International Conference on Open Source Systems Technologies (ICOSST)*, Dec 2015, pp. 170–175.

[7] http://www.ptidej.net/downloads/replications/europlop19/
[8] https://github.com/ResearchML/Catalog-Patterns-MLS

[7] F. Boughanmi, "Multi-language and heterogeneously-licensed software analysis," in *17th Working Conference on Reverse Engineering*, 2010.

[8] D. Galin, *Software quality assurance: from theory to implementation.* Pearson Education India, 2004.

[9] E. Shihab, "Practical software quality prediction," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on.* IEEE, 2014, pp. 639–644.

[10] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[11] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on.* IEEE, 2001, pp. 296–305.

[12] G. Czibula, Z. Marian, and I. G. Czibula, "Detecting software design defects using relational association rule mining," *Knowledge and Information Systems*, vol. 42, no. 3, pp. 545–577, 2015.

[13] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on.* IEEE, 2009, pp. 75–84.

[14] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, Nov 2002, pp. 97–106.

[15] F. Khomh, M. Di Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 75–84.

[16] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 181–190.

[17] S. Liang, *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[18] M. Goedicke and U. Zdun, "Piecemeal legacy migrating with an architectural pattern language: A case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 1, pp. 1–30, 2002.

[19] A. Neitsch, K. Wong, and M. W. Godfrey, "Build system issues in multilanguage software," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 2012, pp. 140–149.

[20] A. Malinova, "Design approaches to wrapping native legacy codes," *Scientific works, Plovdiv University*, vol. 36, pp. 89–100, 2008.

[21] G. Neumann and U. Zdun, "Pattern-based design and implementation of an xml and rdf parser and interpreter: A case study," in *European Conference on Object-Oriented Programming.* Springer, 2002, pp. 392–414.

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

[23] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[24] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 109–118.

[25] A. Osmani, *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide.* " O'Reilly Media, Inc.", 2012.

[26] S. Li and G. Tan, "Finding bugs in exceptional situations of jni programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 442–452.

[27] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, "Traceback: first fault diagnosis by reconstruction of distributed control flow," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 201–212.

[28] P. Mayer and A. Schroeder, "Cross-language code analysis and refactoring," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation.* IEEE, 2012, pp. 94–103.

[29] A. Mouna, K. Foutse, and Y.-G. Guéhéneuc, "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.

[30] A. Mouna, G. Manel, and K. Foutse, "Behind the scenes: Developers' perception of multi-language practices," in *29th Annual International Conference on Computer Science and Software Engineering (CASCON'2019).* ACM, 2019.