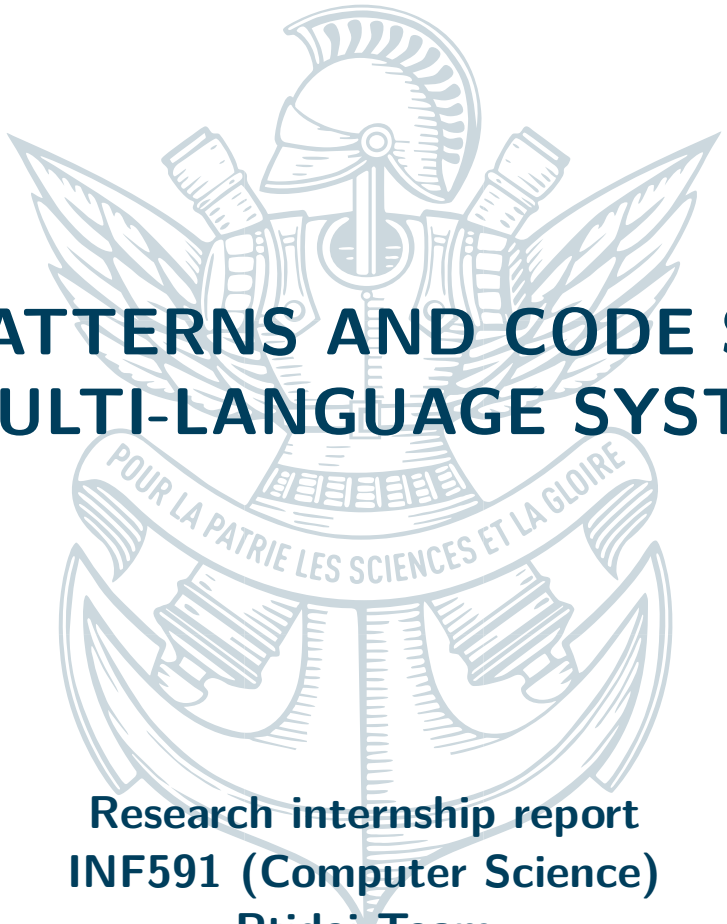

ANTI-PATTERNS AND CODE SMELLS IN MULTI-LANGUAGE SYSTEMS

Research internship report
INF591 (Computer Science)
Ptidej Team

April 1 - August 9, 2019

Palmyra Würtz

DÉCLARATION D'INTÉGRITÉ RELATIVE AU PLAGIAT

Je, soussignée Palmyra Würtz, certifie sur l'honneur :

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport.
3. Que je n'ai utilisé de sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Je déclare que ce travail ne peut être suspecté de plagiat.

Date : 15 août 2019

ABSTRACT

It is now a common practice to program in several languages, in order to use particular features of each of them, or different programming types and paradigms. However, as the number of languages increases, so does the number of design defects. They include anti-patterns at the structural level and code smells at the source code level. They generally do not prevent the program from functioning correctly, but their presence indicates a higher risk of future bugs, and makes the code less readable, thus harder to maintain. That is why, as a research intern in Ptidej lab of Concordia University, Montreal, I tried to figure out an effective way to detect them in programs using Java and C or C++, a language combination that has been widely used and studied. The work is still ongoing but the preliminary results are encouraging.

RÉSUMÉ

Si l'utilisation de plusieurs langages dans un même programme est maintenant chose courante, pour différentes raisons – fonctionnalités particulières ou paradigme de programmation plus adapté –, le nombre de défauts de conception augmente également. Ces défauts incluent les anti-patrons, généralement au niveau de l'architecture du programme, et les *code smells*, caractéristiques localisées au niveau du code source. Ils n'empêchent pas le programme de fonctionner correctement en général, mais peuvent le rendre plus difficilement lisible donc maintenable, et augmentent les risques de survenue de bugs dans le futur. C'est pourquoi, lors de mon stage de recherche dans le laboratoire Ptidej de l'université Concordia, à Montréal, j'ai cherché une technique efficace de les détecter dans des programmes codés en partie en Java et en partie en C ou C++, une combinaison de langage très utilisée et étudiée. Le travail est encore en cours, mais les premiers résultats sont encourageants.

CONTENTS

ACKNOWLEDGMENTS

I would like to thank warmly all those who made this internship possible and enjoyable:

Yann-Gaël Guéhéneuc, my supervisor, who made it possible for me to work in his team in Montreal,

Mouna Abidi, who wrote the article I tried to implement and led my work from wherever she was, whether it be Quebec or Bavaria,

Manel Grichi, who helped me to make the Ptidej software compile and work on all the computers I had to work on,

the Ptidej team, who welcomed me at Concordia University, made me feel at home, let me discover a multicultural environment and whom I left with regret.

LIST OF ACRONYMS

EuroPLoP	European Conference on Pattern Languages of Programs
FFI	foreign function interface
JDK	Java development kit
JNI	Java native interface
JVM	Java virtual machine
PADL	Pattern and Abstract-level Description Language
Ptidej	Pattern Trace Identification, Detection, and Enhancement in Java
MLS	multi-language system
SAD	software architectural defects
SLR	systematic literature review
W3C	World Wide Web Consortium

1 INTRODUCTION

The publication of *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides [3] in 1994, set a precedent in the history of software engineering. The authors, also known as the Gang of Four, defined **software design patterns** as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". Following the publication of this book, other researchers introduced neighbouring concepts, such as traceable patterns, micro patterns [4] and **anti-patterns** [7].

An anti-pattern is 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. As anti-patterns are error-prone and change-prone, it is important to detect and correct them as soon as possible. Maïga [11] drew a parallel between anti-patterns and diseases, quoting Machiavelli's *Prince*: "in its beginning it is easy to cure, but hard to recognize; whereas, after a time, not having been detected and treated at the first, it becomes easy to recognize but impossible to cure" [10]. If the anti-pattern is the disease, then its symptoms are called **code smells**. The latter are defined at a more local level, they are lower-level design defects, while anti-patterns are higher-level design defects.

Although design defects have been thoroughly studied and even though programming in multiple languages has been a wide-spread technique for years, the research about anti-patterns and code smells in multi-language systems is much more recent. Indeed, multi-language systems (also known as heterogeneous, cross-language, multiple languages systems) have many advantages: among them, programmers can choose the language that suits best their needs, and make the most of the benefits of each language; they can reuse a piece of code that is already written in another language. When two languages are used, the main one is called the **host language** and the other one, genereally used because – as a low-level language – it is faster, the **native language**.

My internship took place in the Ptidej lab of Concordia University, Montreal, where I worked specifically with Mouna Abidi. This study is part of a more global project including the definition of anti-patterns and code smells in multi-language systems, their detection and their impact on software quality.

Our research questions are: 1) How to detect design defects in multi-language systems ? 2) Are the detectors efficient (in terms of precision) ?

2 RELATED WORKS

Among the other studies investigating the quality of multi-language systems, we retained particularly those involving the identification of defects.

Issues occurring when building multi-language systems are the topic of Neitsch's study [15], in which the researchers identify patterns and anti-patterns specific to this context. Mayer and Schroeder [12] introduce *Cross Language Links*, by explicitly specifying semantic links, as a solution to suggest refactoring by propagating changes between languages. Kondoh et al. [8] acknowledge that Java

native interface (JNI) is error-prone and focus on four common mistakes developers make with JNI, that led them to write a tool named BEAM to find JNI-specific bugs. The Java development kit (JDK) itself is not free from bugs due to the use of C/C++ in critical sections of the code. Tan et al. [16] discovered six kinds of bugs related to the use of JNI occurring in the JDK. Li et al. [9] warn about JNI exceptions not being handled by the Java virtual machine (JVM), which puts the security of the system at risk. Albeit focusing on JNI, their study can be extended to other foreign function interfaces (FFIs).

3
CODE SMELLS AND ANTI-PATTERNS IN MULTI-LANGUAGE SYSTEMS

After a thorough analysis of documentation, bug reports, a survey and developers' blogs, Abidi introduced a set of six anti-patterns [1] and twelve code smells [2] occurring in multi-language systems. Each one of them is described following a precise template (cf. table 1).

Below is a short summary of the main features of each of these anti-patterns and code smells. Their complete description can be found in Abidi's conference articles for European Conference on Pattern Languages of Programs (EuroPLoP) [2][1].

3.1 Anti-patterns

Excessive inter-language communication A wrong partitioning in both languages leads to many calls in a way or the other. It adds complexity, takes more time to run and may indicate a bad separation of concerns.

Too much clustering of multi-language concerns The multi-language code is concentrated in few classes, regardless of their role and responsibilities.

Too much scattering of multi-language participants Many classes are scarcely used in the multi-language communication.

Unnecessary use of multi-language programming The use of a second language adds complexity but no benefit, e.g. if everything could be achieved as well in a single language.

Language and paradigms mismatch The chosen language is not the best for this task.

Project migration language related issues The program has not been written with evolvability and migration in mind, and it is thus difficult to use new technologies.

[1] Only in the definition of the anti-patterns

Section	Description
Name	The name used to identify the anti-pattern or code smell
Context	The context in which the problem is likely to appear, for example, what the programmer is trying to achieve
Problem	The problem that the anti-pattern or code smell is intended to solve
Supposed Solution	The bad solution that is applied in order to solve the problem
Forces Toward[1]	Reasons leading to apply the supposed solution
Consequences of the Anti-Pattern/Code Smell	Impact of the supposed solution on the code
Forces Away[1]	Reasons leading to refactor or not apply the supposed solution
Refactoring	A better solution that could be applied instead of the bad one
Benefits of the Refactoring	Positive impact of the refactoring, that is to say reasons why the refactoring is a better solution
Related Anti-Patterns[1]	Anti-patterns, not exclusive to multi-language systems, that are related to this one
Related Patterns[1]	Patterns that could possibly be used in the refactoring
Examples	As much as possible, examples taken from real open-source systems; otherwise, minimal examples to illustrate the supposed solution and the refactoring

Table 1: Anti-Pattern and Code Smell Template

3.2 Code smells

Assuming safe multi-language return values A value is returned to the other language without being checked. Thus, the interaction between both languages may be ill-performed.

Hard coding libraries When different libraries are needed depending on the OS, they are not loaded with conditions on the OS, but for instance with a try-catch mechanism, making it hard to know which library have really been loaded.

Local references abuse The developer does not manage the memory in the native space properly, and does not release local and global references.

Memory management mismatch Reference types passed from one language to another are not released in a language which does not handle the management of memory causing memory leaks.

Not caching objects' elements A method is called to retrieve a field every time this field is needed, although the field's ID or value could have been cached.

Not handling exceptions across languages The exceptions are not handled, because the developer generally relies on the exceptions provided by the other language.

Not securing libraries The code loads a foreign library without any security check or restriction.

Not using relative path to load the library A library is loaded thanks to its name only, and not its path. It cannot be accessed in the same way from everywhere.

Passing excessive objects A whole object is passed as an argument, although only some of its fields were needed, and it would have been better for the system performance to pass only these fields.

Unused native method declaration A method is declared in the host language but not implemented in the foreign language.

Unused native method implementation A method is declared in the host language and implemented in the foreign language, but never called from the host language.

Unnecessary parameters Some arguments of a function are used neither in its body nor in the other language.

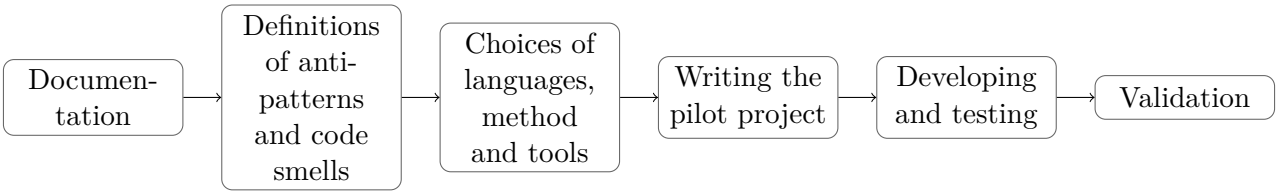

Figure 1: Development of the tool

4 IMPLEMENTATION

The figure 1 shows the steps we followed to develop our detection tool.

4.1 Choice of the languages

According to the systematic literature review (SLR) of Abidi, the most popular combination of languages studied in papers related to multi-language systems is the set Java and C or C++. The interaction between languages is provided by JNI in most cases. JNI lets Java code call and be called by native code, that is to say C or C++ code. Throughout the rest of this report, we will refer to both C and C++ as "C/C++", because C++ is an extension of C and JNI applies to both of them (although the syntax is a little different).

That is why we decided to code a program able to detect occurrences of anti-patterns and code smells in systems using Java and C or C++ interacting through JNI.

4.2 Choice of the anti-patterns and code smells

Among the listed anti-patterns and code smells, we chose a subset that seems possible to detect automatically, namely all the code smells, and the following anti-patterns: *Excessive Inter-language Communication*, *Too Much Scattering of Multi-language Participants* and *Too Much Clustering of Multi-language Concerns* (cf. table 2). Indeed, the remaining three anti-patterns depend on subjective or hard-to-measure features, since for *Language and Paradigms Mismatch*, for example, we should be able to understand the developer's intention and to choose the best language for that.

The defects are described in a general way, without any details on how to detect them precisely, in the article. Actually, what they look like depends on the languages in which they appear. We had to understand the form they take in Java and C/C++ and define measurable properties to detect them algorithmically. The numeric values of these properties are generally taken from developers' resources.

4.3 Choice of the method

The first idea was to use the Pattern and Abstract-level Description Language (PADL) meta-model developed by Guéhéneuc [5]. However, we found some drawbacks: this meta-model is defined at the

Anti-patterns	Code smells
Excessive inter-language communication	Assuming safe multi-language return values
Too much clustering of multi-language concerns	Hard coding libraries
Too much scattering of multi-language participants	Local references abuse
Unnecessary use of multi-language programming	Memory management mismatch
Language and paradigms mismatch	Not caching objects' elements
Project migration language related issues	Not handling exceptions across languages
	Not securing libraries
	Not using relative path to load the library
	Passing excessive objects
	Unused native method declaration
	Unused native method implementation
	Unnecessary parameters

Table 2: List of anti-patterns and code smells in multi-language systems. The grayed cells are the design defects whose detectors have been implemented.

class and interface level, we should have extended it to a lower level to deal with variables; detecting some code smells requires to know if some variables are used in a loop and other details that such an abstraction may not be able to convey; finally, according to Guéhéneuc, the parser used for C/C++ has major flaws, which make it inadequate for our project.

We also considered the possibility to use rule cards and constraint programming to generate the code of the detectors, like the design pattern identifier DeMIMA [6]. Nevertheless, as Moha stated, "constraint programming is not adapted for the detection of defects" [13].

Rejecting PADL, we decided to use srcML [2], a parsing tool that converts source code into srcML, which is an XML format. An interesting feature of srcML is that it can take a directory or several files as an input and output only one srcML file. It is helpful, since some anti-patterns and each code smells involve several files, for example all the classes of a package or a Java class and the native functions it uses. The conversion of source code into the srcML format is reversible: once the srcML tags are removed, the result is identical to the source code. The srcML command has an `--position` which creates new tags and attributes giving the line and column numbers. It is relevant to access the exact position of a design defect we found. However, its use dramatically increases the size of the srcML result, thence the execution time of the detection. Thus, we do not use it in a first time. Once an anti-pattern or a code smell is detected, the detector also finds its file, package, class, method and variable name (if applicable) to locate as precisely as possible.

```
public class HelloWorld {

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

}
```

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

Listing 1: Example of Java code

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

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

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

Listing 2: Code converted to srcML

The code can then be explored thanks to XPath queries, as any XML. XPath is a query language developed by the World Wide Web Consortium (W3C) that can select nodes in an XML tree.

```
parameter_list/parameter[position()>2]/decl[type/name = 'jobject']/name
```

Listing 3: Example of XPath query, that selects the name of an argument whose type is `jobject` and which is neither the first nor the second argument

4.4 MLS SAD, our detection tool

The detection tool we developed is coded in Java. We chose this language for several reasons: it is the main language of the Ptidej tool suite; it has several libraries to handle xPath queries; we already had experience in this language.

The first step was to create a pilot project[3] with simple examples of these anti-patterns and code smells, and with a "clean code", and to write unit tests running on this code. Then, a detector was created for each anti-pattern and each code smell. These detectors navigate through the generated srcML file, exploring the nodes to find the appropriate properties, and return a set of the defect. The class **DetectCodeSmellsAndAntiPatterns** runs all the detectors on the file or directory given as an input, and outputs a CSV file with the results (cf. fig. 2).

[3] `https://github.com/ptidejteam/PilotProjectAPCSMLS/`

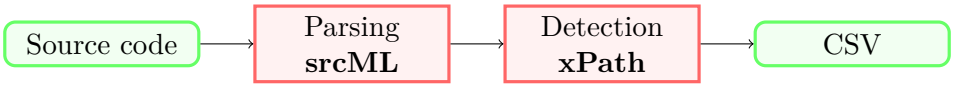

Figure 2: Analysis flow

Our implementation, called MLS SAD[4] (multi-language system software architectural defects) was first planned to be part of the Ptidej tool suite, a software designed to find design patterns and design defects in programs. It could then have been a generalization of Ptidej's SAD tool for multi-language systems. Contrary to the others tools of Ptidej, MLS SAD does not use the PADL meta-model. That is the reason why we decided not to integrate it into the suite immediately.

5 VALIDATION PROCESS

5.1 Systems

We chose 10 systems depending on the following criteria:

1. they are open source
2. their source code is partly written in Java and in C/C++, and uses JNI
3. their repositories are active (in terms of commits).

The selected systems are taken from different fields (cryptography, video game engine, messaging app, database management, etc.), some of them being well-known, in order to check the hypothesis that the anti-patterns and code smells defined above are spread across all sorts of code. Java, C and C++ do not have to be their main language. Even if their proportion is low in a system, the high number of lines of code implies that there are many Java and C/C++ code lines. The selected systems are to be found in table 3.

5.2 Precision and recall

The ongoing work consists in detecting anti-patterns and code smells in these systems with our program on the one hand and manually on the other hand in order to compute the precision and recall of our detectors. These values are not expected to be very high for the following reasons:

- *Precision:* Deciding whether a piece of code is a design defect sometimes depends on the context and the intention of the programmer.
- *Recall:* We voluntarily let some cases apart for the moment, for example the ones that would require more specific tools than XPath queries.

[4] `https://github.com/PalmyreB/MLS-SAD/`

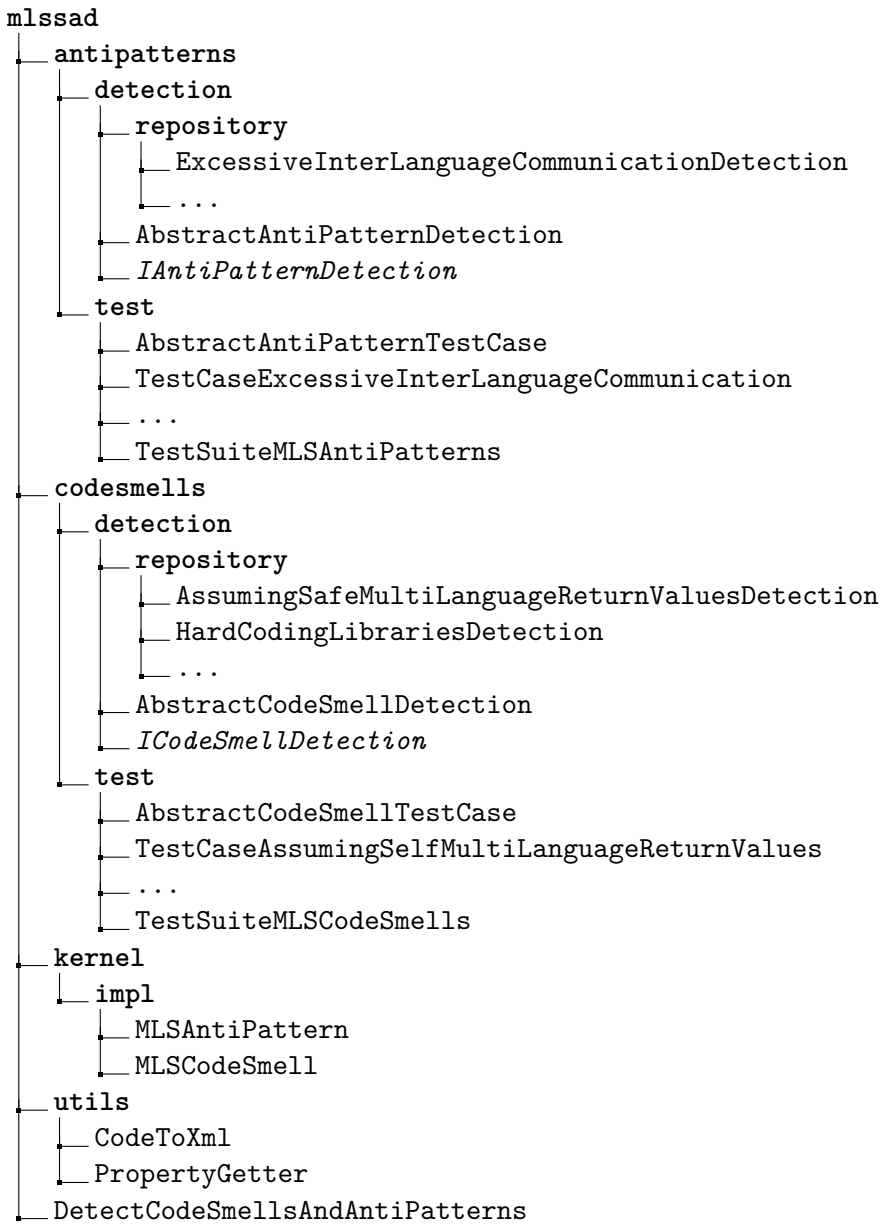

Figure 3: Directory tree of the **mlsad** package with **packages**, classes and *interfaces*

Name	Field	Number of code lines	% Java	% C/C++
Conscrypt	Cryptography (security provider)	64.9K	85.2	14.1
PL/Java	Database management	42.6K	64.9	28.8
OpenJ9	Java Virtual Machine	1.5M	35.7	52
jMonkey Engine	Game engine	838K	38.6	1
Telegram	Messaging client	1.13M	36	53.1
JavaSMT	Computation	28.6K	87.9	4.4
JNA	Programming	258K	71.4	15.8
TensorFlow	Computation	2.3M	0.7	54.2
RocksDB	Database management	332K	10.7	83
react-native	Programming	364K	17.4	16.5

Table 3: List of selected systems[5]

System	Precision	Recall
Conscrypt	96.7 %	83 %
PL/Java	99.2 %	
OpenJ9	> 96 %[6]	

Table 4: Precision and recall of some systems

The results are still scarce, because of the lack of time and resources. However, the first precision results are very promising, since the precision is always higher than 90 % (cf. table 4).

6 LIMITS AND FURTHER WORKS

Our work has some limits and threats to validity:

- The list of design defects is not claimed to be exhaustive and is likely to evolve in the future.
- As previously mentioned, our detection tool does not address all the anti-patterns cataloged above, and not even all the cases of the chosen design defects.
- The values of the measurable properties are based on developers' resources and feedback, but are not objective values. The user of our tool can modify them for its use in a configuration file.
- We are conscious of the limits of hard-coding the detection of smells [14], but it seemed the most adapted solution in our case.
- Our tool is not adapted for very large systems, with more than a million lines of code. A partial solution could be to detect the appropriate defects on small parts of the system first, then only the other defects on the whole system. We call "appropriate defects" here the defects that involve

[6]The precision has been computed for each part of the system separately, but not yet for the system as a whole. Thereby, only the precision for design defects involving a single file is given here.

only a single file. Another solution would be to write the srcML into a file, then read it part by part, but that would jeopardize the speed of the detector.

- The location may not be that helpful if the involved method is overloaded. That is to say, if two methods bearing the same name exist at the same location (file, package, class), the developer will have to check manually which one is involved in a defect. There are several ways to avoid this problem: either by using the line numbers, as discussed above, or by getting the signature of the method (but then the developer would still have to check the signature of each method bearing the same name).
- In our results, some design defects appear much more frequently than others (for example, *Unused parameters* often occurs, because of classes complying with an interface). The set of those who appear rarely may be too small to know whether our tool is efficient to detect them.

The next step will be to finish to compute the precision and recall on the selected systems, to validate or not our tool. Then, we could still improve it and address some of its limits. We should also compute the precision and recall for each anti-pattern and code smell, to improve the matching detectors. Finally, if the tool is validated, we could measure the impact of the cataloged design defects on the quality of multi-language systems.

In the future, this work should be extended to other language combinations, such as Python/C, a popular combination among developers, according to Abidi's SLR. An automatic correction could also be considered for some of the anti-pattern or code smell.

7 CONCLUSION

In this report, we talked about design defects that were not yet listed as such, or whose description was less complete in previous articles, about how to detect them in software developed with Java and either C or C++, and we discussed about the performance of our method. Design defects are known to lessen the maintainability of systems and make them more error-prone and change-prone, thence our research.

In a nutshell, although this study is still in progress and our tool has room for improvement, we hope to prove that it is possible to detect automatically most of the anti-patterns and code smells of multi-language systems. In a near future, developers could use it to assess the quality of their code and correct it at an early stage, to avoid bugs and higher maintenance costs.

REFERENCES

[1] Mouna Abidi et al. "Anti-Patterns for Multi-language Systems". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP)*. July 2019.

[2] Mouna Abidi et al. "Code Smells for Multi-language Systems". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP)*. July 2019.

[3] Erich Gamma et al. *Design Patterns. Elements of reusable Object-Oriented Software*. Prentice Hall, Dec. 1, 1995. ISBN: 0201633612. URL: https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html.

[4] Joseph (Yossi) Gil and Itay Maman. "Micro Patterns in Java Code". In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 97–116. ISSN: 0362-1340. DOI: 10.1145/1103845.1094819.

[5] Yann-Gaël Guéhéneuc. "Ptidej: Promoting patterns with patterns". In: *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Ed. by Springer-Verlag. 2005.

[6] Yann-Gaël Guéhéneuc and Giuliano Antoniol. "DeMIMA: A Multilayered Approach for Design Pattern Identification". In: *IEEE Transactions on Software Engineering* 34.5 (Sept. 2008), pp. 667–684. DOI: 10.1109/tse.2008.48.

[7] Andrew Koenig. "Patterns and Antipatterns". In: *Journal of Object-Oriented Programming* (1995).

[8] Goh Kondoh and Tamiya Onodera. "Finding bugs in Java native interface programs". In: Jan. 2008, pp. 109–118. DOI: 10.1145/1390630.1390645.

[9] Siliang Li and Gang Tan. "Finding bugs in exceptional situations of JNI programs". In: Nov. 2009, pp. 442–452. DOI: 10.1145/1653662.1653716.

[10] Niccolò Machiavelli and William K. Marriott. *The Prince*. J.M. Dent, 1916.

[11] Abdou Maïga. *Détection et correction automatique des défauts de conception au moyen de l'apprentissage automatique pour l'amélioration de la qualité des systèmes*. Tech. rep. EPM-RT-2010-12. Polytechnique Montréal, 2010. URL: https://publications.polymtl.ca/2635/1/EPM-RT-2010-12_Ma%5C%C3%5C%AFga.pdf.

[12] Philip Mayer and Andreas Schroeder. "Cross-Language Code Analysis and Refactoring". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Sept. 2012, pp. 94–103. DOI: 10.1109/SCAM.2012.11.

[13] Naouel Moha. "DECOR: Détection et correction des défauts dans les systèmes orientés objet". PhD thesis. Université de Montréal, 2008.

[14] Naouel Moha et al. "From a domain analysis to the specification and detection of code and design smells". In: *Formal Aspects of Computing* 22.3 (May 2009), pp. 345–361. DOI: 10.1007/s00165-009-0115-x.

[15] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. "Build system issues in multilanguage software". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sept. 2012, pp. 140–149. ISBN: 978-1-4673-2313-0. DOI: 10.1109/ICSM.2012.6405265.

[16] Gang Tan and Jason Croft. "An Empirical Security Study of the Native Code in the JDK". In: Jan. 2008, pp. 365–378.