Anti-Patterns and Code Smells for Multi-language Systems

Mouna Abidi[1], Manel Grichi[1], Yann-Gaël Guéhéneuc[2], and Foutse Khomh[1]

[1] Polytechnique Montreal, Canada
[2] Concordia University, Canada. mouna.abidi@polymtl.ca

Abstract. Multi-language systems are common nowadays because most of the systems are developed using components written in different programming languages. These systems could arise from three different reasons: (1) to leverage the strengths and take benefits of each language, (2) to reduce the cost by reusing code written in other languages, (3) to include and accommodate legacy code. However, they also introduce additional challenges, including the increase in the complexity and the need for proper interfaces and interactions between the different languages. To address these challenges, the software-engineering research community, as well as the industry, should describe and provide common guidelines, idioms, and patterns to support the development, maintenance, and evolution of these systems. These patterns are an effective means of improving the quality of multi-language systems. They capture good practices to adopt and bad practices to avoid. We analysed open-source systems, developers' documentation, bug reports, and programming language specifications to extract bad practices of multi-language systems usage. We encoded and cataloged these practices in the form of code smells and design anti-patterns. We report in this paper six anti-patterns and 12 code smells.

Keywords: Anti-patterns, code smells, multi-language systems

1 Introduction

The quality of software systems become increasingly important with the evolution of technology, the high demand for software systems by society, and market competition. Providing systems with good quality is a necessity and no longer an advantage [1]. Software quality is one of the most important concerns for systems to reduce testings, maintenance, and evolution costs [2,3].

Software quality partly depends on adopting guidelines, idioms, patterns, and avoiding code smells and design anti-patterns. For example, design patterns [4] describe good solutions to recurring design problems. On the contrary, design anti-patterns describe poor solutions to design problems [5,6]. Design anti-patterns are negative practices at the design level, while code smells are at the implementation level. Many studies reported that the use of design patterns improves software quality while code smells and design anti-patterns negatively

impact software quality [7,8,9]. Developers are recommended to remove anti-patterns and code smells as soon as possible through refactorings, which are behavior preserving code transformations.

Most non-trivial software systems are developed using more than one programming languages. These systems could arise from three different reasons: (1) to leverage the strengths and take benefits of each language, (2) to reduce the cost by reusing code written in other languages, (3) to integrate systems originally written in different programming languages, e.g., while integrating systems from two different companies. Developers can reuse existing modules and components, without writing the source code from scratch [10]. They choose the programming language most suitable for their needs, instead of implementing all the tasks with a single language [11,12,13].

Several studies in the literature discussed multi-language systems, with studies mostly reporting the importance of addressing the complexity as one of the main challenges of multi-language systems [14,15]. Some of the studies highlighted the importance of investigating multi-language systems design patterns and anti-patterns [16]. Tan et al. [10] introduced a taxonomy of bugs that may result from combining Java and C++. Goedicke et al. [17] proposed a pattern-based approach to wrap legacy components as black-box entities. They introduced five architectural patterns based on well known design patterns [18]. These patterns are proposed to wrap a system at different granularity levels, including wrapping an existing C implementation into an object system. Neitsch et al. [16] investigated build issues and defined build patterns and anti-patterns to help developers building multi-language systems.

To support the software quality of multi-language systems, we extracted, encoded, and cataloged good and bad practices in the development, maintenance, and evolution of multi-language systems. We analysed the source code of open-source multi-language systems as well as developers' documentation, and programming-language specifications. These systems contain mainly Java/C(++) but also include other programming languages e.g. Python, JavaScript, Lua, etc. We observed good and bad practices in the code as well as issues reported in the developers' documentation and bug reports. We encoded and cataloged these observed practices and report our findings in the form of anti-patterns and code smells. These anti-patterns and code smells could apply to microservices or, rather, to the implementation of microservices, as to any other pieces of code in which such poor design or implementation choice could appear. We have focused our efforts on design anti-patterns and code smells to complement the previous work on identifying design patterns for multi-language systems [17,16,19,20]. Both design patterns and anti-patterns are important to improve the quality of multi-languages systems and help developers coping with their challenges. We believe that our results could help not only researchers but also developers, maintainers of multi-language systems, and also any of those considering using more than one programming language in the same project.

We also outline some observations, some of the reported anti-patterns and code smells are related to these observations: (1) The choice of the programming

language may directly impact the performance depending on the programming tasks; (2) A specific programming language may present some advantages that are not guaranteed with another language; (3) The difference in features and incompatibility of the languages can lead to correctness problems; (4) Usually the system will be maintained by a different person. We described the consequences of these code smells and anti-patterns based on logic and observations and seek the community's feedback. In future work, we will extend this research and measure concretely the impact of these negative consequences on software quality.

The remainder of this paper is organised as follows: Section 2 discusses the background of multi-language systems and related works. Section 3 describes our methodology for gathering good and bad practices. Section 4 reports multi-language systems Anti-patterns, while Section 5 reports multi-language systems code smells. Section 6 summarises threats to the validity of our methodology. Section 7 concludes the paper and discusses future works.

2 Background and Related Work

We now present a brief background about multi-language systems, patterns, and anti-patterns, in general. We then discuss some related works.

Multi-language Systems: Developed using more than one programming language. Most of the systems with which we interact daily integrate components written in several, different programming languages. Developers of these systems attempt to choose the "right" programming language for each component. The adjective "right" covers every choice, from using the most appropriate programming language instead of trying to solve all problems with a single language [15], to using the programming language that some particular developers know best. The resulting heterogeneous components usually communicate through Foreign Function Interfaces (FFIs) [21]. Some of the multi-language systems also rely on language binding that are wrapper libraries offering a bridge between two programming languages, so that a library written for one language can be used in another language.

Patterns and Anti-patterns: Patterns were introduced for the first time in the domain of architecture by Alexander [4]. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, In such a way that you can use this solution a million times over, without ever doing it the same way twice" [4, p267]. From architecture, design patterns were introduced in software engineering by Gamma et al [18]. In their landmark book, Coplien et al. [22] provided an overview of practical guidelines for design pattern usage. They presented design patterns as a means to meet the goal of capturing the design of complex object-oriented systems. These design patterns are based on the developers' experiences when facing recurrent problems and applying "good" solutions to solve these problems.

The goal of encoding and cataloging design patterns is to preserve, share, reuse, and improve design knowledge and take the benefits from similar, past situations [22,23].

Anti-patterns are "opposite" to design patterns. They document "poor" solutions to recurring problems [5,6]. In the literature, there are two main types of such bad practices: anti-patterns and code smells [24,6,5]. Several studies showed that the presence of anti-patterns makes the evolution of the software more difficult. They affect software comprehensibility and increase change- and fault-proneness and increase the effort needed to perform maintenance activities [25,26]. For example, classes including design defects are significantly more fault-prone and change-prone compared to classes without those occurrences [27,28].

Encoding and Cataloguing: There exist several templates in the literature to encode patterns and anti-patterns. The template that we used in this paper is inspired by the *WikiWikiWeb*[3]. We adapted the template to the specificity of our work as follows:

- Anti-pattern/Code Smells: We describe in the title the name to identify the anti-pattern or the code smell.
- Context: The context in which this particular anti-pattern or code smell applies, for example, real-time systems or communication systems.
- Problem: It introduces the initial problem that is being solved or the problem that may lead to the wrong solution. It can be illustrated by a simple concrete example.
- Supposed Solution: The supposed solution is the solution solving the problem on first thought but that has other negative impacts on software quality.
- Forces Toward: Used in the anti-pattern section, they describe common reasons and choices that may lead to the application of the bad solution. We relied on the forces discussed in the anti-pattern book to write the forces [5]. They are based on the Software Design-Level Model (SDM) that are the general forces ignored, misused, or overused in the Anti-pattern. They can also be contextual motivating factors that influence design choices.
- Consequences of the Anti-Pattern: These consequences describe the impact of applying the "poor" solution to solve the problem.
- Forces Away: Used in the anti-pattern section, the forces away provide the decisions and reasons to avoid the bad solution. Similar to the *Forces Toward*, these forces are inspired by Brown's book and can be related to the management of functionality, performance, complexity, changes, resources, and technology transfer [5].
- Refactoring: This solution (or solutions) presents the better solution that can be applied to remove the anti-patterns or code smells. It includes the steps that can be followed to apply the solution.
- Benefits of the Refactoring: These benefits are the consequences describing the positive impact of applying the refactoring to remove the occurrences of the code smells and–or anti-patterns.

[3] `http://wiki.c2.com/`

- Related Anti-Patterns: If any, specify the names of related code smells and–or anti-patterns.
- Related Patterns: If any, specify the names of patterns that could be used in the refactoring. In this paper, we specify the names of the patterns that could be applied in the refactored solution. In future work, we will examine the effectiveness of those solutions.
- Examples: These examples provide code and–or diagrams showing the anti-pattern or code smell in context. When possible, the example is taken from real systems or is a Minimal, Complete, and Verifiable example. In some cases, we provided a small fictive example. In other cases, we did not add examples, especially where, the anti-pattern or code smell seems evident, or cannot be well illustrated with only one example. We used a detailed description to illustrate the situation.

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

Neitsch et al. [16] studied five multi-language software packages from Ubuntu 9.10. They provided common build patterns and anti-patterns that summarise the key problems related to the build of multi-language systems.

Goedicke et al. [17] proposed five architectural patterns based on well known design patterns. These patterns are defined to wrap legacy components as black-box entities. Most of the defined patterns can be used with different programming languages. To assess the legacy migration and the wrapping techniques, the authors also presented a pilot project. They also provided a detailed definition of these patterns. The pattern *Object System Layer* provides a highly flexible object system as a layer build on top of a given language [29]. It makes components that are not object-oriented or that are implemented in another language, accessible through *Object System Layer*. These components can then be treated as black-boxes. The pattern *Message Redirector* ensures a simple indirection architecture that maps the calls to a message implementation [30]. It also provides callback methods around the calls.

Malinova [19] made an attempt to connect some well known design patterns e.g. Adapter, Proxy, and Wrapper Facade, to the process of Java wrapping of native legacy codes. In this paper, design patterns were studied in the context of invoking native applications from Java code.

Kondoh et al. [31] focused on four kinds of common JNI mistakes made by developers. They proposed BEAM, a static-analysis tool to find mistakes pertaining to error checking, virtual machine resources, invalid local references, and JNI methods in critical code sections. They did not propose recommendations to avoid and–or fix these mistakes.

Osmani et al. [32] presented the Lazy Initialisation pattern which describes how to execute Ajax requests in JavaScript, where the Ajax request includes a URL and some data, possibly in JSON or XML, to communicate with a server, likely implemented in C/C++.

Li and Tan [33] highlighted the risks caused by the exception mechanisms in Java, which can lead to failures in JNI implementation functions and affect secu-

rity. They defined a pattern of mishandling JNI exceptions. This paper focused mainly on JNI but can also be adapted to other FFIs, such as the Python/C and the OCaml/C interface.

Tan et al. [10] studied the JNI usages in the JDK source code. They examined a range of bug patterns in the native code and they identified six bugs related to the use of JNI methods in the JDK. Bugs identified can cause a JVM crash or can open the JVM to some security breaches. They found that bugs are possible due to language mismatches and the assumptions made by the Java code regarding the C(++) code. As an example, the native method *java.util.zip.Deflater.deflatesByte()* assumes that its Java callers check bounds, which could lead to buffer overflows.

Ayers et al. [34] proposed TraceBack a tool that collects and analyses bugs in multi-languages systems by storing data through runtime instrumentation of control-flow blocks. They collected the data by statically rewriting the libraries and-or instrumenting the intermediate languages to generate a unified trace of components' execution.

Mayer and Schroeder [35] studied the dependencies in multi-language systems. They proposed a technique to identify dependencies among multi-languages components, warn of potential missing dependencies, and propagate renaming among multi-language code.

3 Study Design

In this section, we detail the setup of our study. We present the steps followed to collect the anti-patterns and code smells. Figure 1 presents an overview of our methodology. We believe that the following steps could be used for a replication purpose as well as for any future study investigating new design patterns and anti-patterns.

Prerequisites: We believe that to investigate and collect good practices, design patterns, idioms— and bad practices—design anti-patterns and code smells for multi-language systems, it is important to have enough knowledge and experience with both multi-language systems and design patterns and anti-patterns. From previous studies and our literature review, we already had good background and knowledge on design patterns, design anti-patterns, and code smells for mono-language systems. Two of the authors of this paper have also experience in developing tools to detect occurrences of design patterns, design anti-patterns, and code smells.We performed in a prior study a systematic literature review and a practice review to investigate and compare the usage of multi-language systems in the literature and in real systems hosted in GitHub. This step gave us good knowledge about multi-language systems and their challenges as well as design patterns and anti-patterns. It helped us to collect some keywords that can be used to retrieve challenges and issues related to multi-language systems. It also allowed us to better distinguish between a simple habit and a possible pattern or code smells.

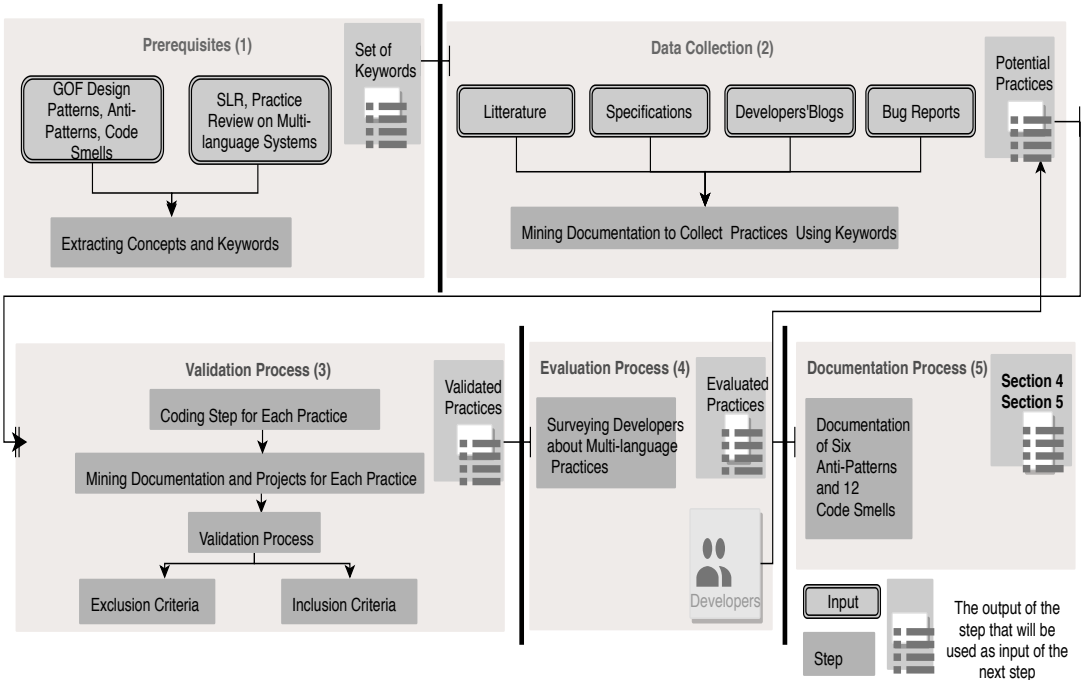

Fig. 1. Overview of the methodology used to collect and document the anti-patterns and code smells for multi-language systems.

Data Collection: Once we decided to collect and document anti-patterns and code smells for multi-language systems. We started by mining all possible sources of documentation. We searched in the literature, language specification, developers' blog as well as bug reports. From our systematic literature review on multi-language systems. We found that the most studied combination of languages is Java/C(++). For that, We decided to start with this combination and then include other languages. We deeply read the Java Native Interface specification [36] as well as developers' documentation to collect common practices and guidelines related to the JNI and multi-language systems. We searched in Google as well for JNI practices and find a couple of developers' blog and documentation that discuss common good and bad practices[5] [4]. We documented the practices extracted in terms of definition, context, and examples.

We then considered multi-language systems in general and searched for any other possible issues related to a combination of more than one programming languages. We analysed bug reports and developers' documentation to extract the issues related to multi-language systems that have been reported by developers. We relied on often used websites such us *Stack Overflow*, *GitHub issues*,

Bugzilla, *IBM Developers*[4], and *developer.android*[5]. We queried the developers' documentation and bug reports by searching for common keywords that reflect issues in multi-language systems. We relied on some keywords collected from our literature review as well as common issues reported by developers. We used the set of keywords extracted from the previous step including *JNI issue, Python/C issue, foreign library, API, polyglot, programming languages issues, incompatibility, compilation errors, memory issues, performance issues, security issues, foreign function interface*. As an example, when searching for *JNI issue* in *Bugzilla*, our query returned 23 results, among them we considered only two as possible bad practice. One was related to the library loading, the other was related to the management of exceptions[6] [7]. From *Stack Overflow*, for the keywords *JNI issue* and *Python/C issue*, we had for each of these keywords 500 results, we searched manually only for issues that have been already discussed in the developers' documentation[5] [4].

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

Data Validation: For each of the practices reported in our list of potential practices, we performed a coding process in which, we provided a definition and explanation of the practice. The explanation was in term of what are the contexts, situations, and possible examples that we should look for to retrieve occurrences of the practices. We performed a discussion between the authors to validate the explanations provided for the potential practices. We performed the validation process through different sources of information following inclusion and exclusion criteria. Through this step, we aimed to verify if the potential reported practices have been used or discussed in at least three situations and-or examples in open source systems. We searched for occurrences of these practices in different sources of information (e.g. GitHub, Developers' Blog, Bug Report).

We defined a set of inclusion and exclusion criteria. As inclusion criteria, we considered a practice that was discussed in at least three situations. In the case of literature or any other type of documentation, we searched for similar situations that have been reported by developers, discussed in bug reports or developers' blog. Another inclusion criteria were when analysing the source code of multi-language systems. We searched if the good or bad practices discussed in

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

the literature were present in at least three classes, source code files, or systems. We also considered the case where that practice was discussed as a good one but was not followed in some open source systems (e.g. The code smell *Not Checking Exceptions* that was discussed in developers' documentation[4] and in some articles but was not followed in most of the systems that we analysed). As exclusion criteria, we considered a practice for which, we were not able to find at least three of its occurrences in any of the sources of information, including open source systems. We also considered as exclusion criteria practices that seem more likely to be a simple habit than a potential code smell or anti-pattern.

We manually searched in the source of information (e.g. bug reports, developers' blog, developers' documentation) to find at least three situations where the potential practices were discussed. We also used data already extracted from one of our prior studies focusing on JNI usage. The data consists of 100 multi-language open source systems. We extracted these systems from OpenHub using Python Scripts. We then downloaded the projects and manually analysed their source code. These systems were mainly JNI systems but also contain other languages. OpenHub provided the list of all the languages used in the project. These systems contained not only Java/C(++) but also Python, JavaScript, Lua, etc. (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). Here are some systems in which we mainly focused more during this study: *libgdx, Google toolkit, Openj9, Rocksdb, JMonkeyEng, OpenVRML, PortAudio Java Bindings, jpostal, JavaSMT, Jna, ZMQ, reactNative, Telegram, OpenCV, Tenserflow, JatoVM, SQLlite, Frostwire, Godot, python-telegram-bot.* We provide in sections 4 and 5 the sources and-or name of the projects from which we extracted the code smells and-or anti-pattern.

We manually and qualitatively analysed the source code of the multi-language systems collected from GitHub to extract occurrences of good and/or bad practices. We also checked if the common guidelines and practices reported in the literature are followed by the developers in practice. Most of them were not, in that case, we reported it as a possible bad practice. In our case, we considered as multi-language systems practices a piece of code that is involved in the multi-language systems and participating in the interaction between two or more languages and that has been documented in the literature as bad practice or that have been reported in bug reports or developers' documentation as causing issues or negatively impacting the system. We considered as practices a similar situation and-or that were observed more than three times and was discussed in the programming language specification, or developers documentation as being a wise practice.

Data Evaluation: In order to evaluate our set of anti-patterns and code smells, we asked in our survey on multi-language systems if developers have faced these practices. We also asked them in this survey about any good or bad practices that they are adopting or avoiding when using multi-language programming. We added the proposed good and bad practices in our list of possible practices so they can be used in future work as input to the validation process. We will

use this survey for future work to investigate the challenges related to multi-language systems as well as the practices used by developers to cope with those challenges.

Anti-patterns and Code Smells Documentation: We reported all the observed practices and performed a discussion between the authors to validate if the practices are really valuable and should be documented in form of anti-patterns or code smells, or if they are only a simple practice or developers' habit. We discussed each case until a consensus was reached. We used an available template to documented our results in the form of six anti-patterns and 12 code smells as presented in section 2. We believe that these practices could help researchers and developers to cope with the challenges introduced by multi-language systems. In future work, we will investigate more practices and document them in the form of design patterns, anti-patterns, and code smells. The figure 2 presents a pattern overview of the collected anti-patterns and code smells and the relationships between them.

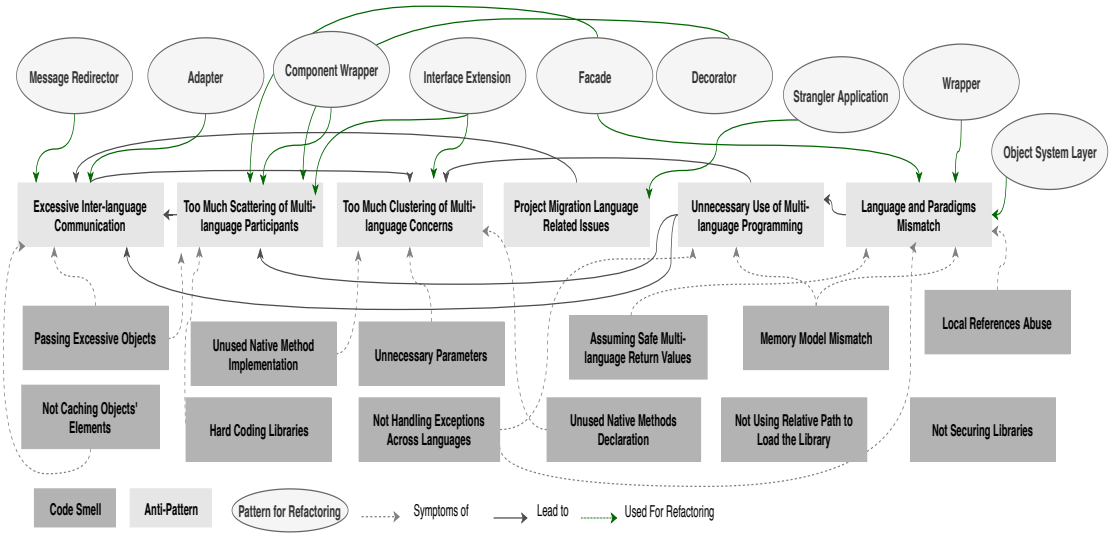

Fig. 2. Pattern Overview Diagram

4 Anti-Patterns for Multi-language Systems

In this section, we present a catalog of the extracted good and bad practices in the form of anti-patterns following the template detailed in section 2.

4.1 Excessive Inter-language Communication

Context: Supposing we are in a context in which we must implement a task or add a new feature that is already available as library implemented in another language. This can also be illustrated with a situation when a single language is not suitable to implement all the tasks. It may appear in the context of embedded systems, or systems in which we need an important number of communication between different layers or modules of the application.

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

Supposed Solution: Connect existing modules that are implemented using different languages and/or technologies. Reuse existing codes or modules implemented in different languages to benefit from the reuse of code or it can simply be related to the fact that some tasks are easier implemented in a specific language or are already available and ready to be used. The bad solution would be, to add the foreign code and access features from one language to another each time in the program we need to access foreign objects without considering the number of calls from one language to another.

Forces Toward: (1) Wrong partitioning of parts in the languages; (2) Trying to benefit of performance of another lower-level language; (3) Using a scripting language to enable non-programmers to participate; (4) Providing several wrappers to access the features of the system; (5) Not separating all the multi-language concerns; (6) During a change or new requirement, design decision tends to introduce several calls rather than to refactor working code; (7) Classes designed with high coupling.

Consequences of the Anti-Pattern: This anti-pattern will result in an excessive passage of objects and calls between the host and the foreign language. In a study focusing on JNI systems, they found that calling the native code from Java code can take five times longer than a regular method call[4]. Similarly, calling Java code from the native code can take substantial time. If the partitioning of tasks between the foreign and the host languages is not used properly, this can cause a dispersion of the responsibility to perform a simple task between several languages. In some cases we can have excessive calls and passage of parameters between one language to another, These calls add more complexity to the program and negatively impact the first reported observation related to the performance of the system that we presented in section 1.

Forces Away: (1) Design components with high cohesion; (2) Separate the concerns; (3) Ensure efficiency and management of the resources by limiting the number of methods calls and messages sent between components; (4) Information hiding and avoid indecent exposition.

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

Benefits of the Refactoring: Refactor this anti-pattern ensure high cohesion and low coupling. A better performance by reducing the number of calls from one language to another. It also reduces the complexity, by limiting and splitting the responsibility between the host and the foreign code. Another benefit is to avoid unnecessary broken code related to a nonseparation of concerns when applying changes.

Related Anti-Patterns: Circular dependency.

Related Patterns: Message Redirector [30] and Adapter[22].

Examples: Occurrences of this anti-pattern are generally observed in systems involving different layers or components. For example, the same object can be used and-or modified in more than one modules written in different languages. Each time we need the object, we pass it from one language to another or we call the foreign method to perform specific tasks in the object. The solution would be to separate the responsibilities and minimise the calls between the languages. It is better to focus each time in a single language to implement the tasks. Some examples of this anti-pattern have been observed in *Godot, PortAudio Java Bindings, OpenResty.* In *Godot*, The function *process()* is called at each time delta. The time delta is a small period of time that the game does not process anything i.e. the engine does other things than game logic out of this time range. The foreign function *process()* is called multiple times per second, in this case once per frame[8]. Another example in *PortAudio Java Bindings*, where they used raw buffers between Java and C++. In this example, they copy data between buffers which makes way for more communication than needed. Java supports memory mapped input/output for this purpose, with this raw buffers can be used between language barriers[9]. We present in figure 3, an example of

[8] `https://github.com/godotengine/godot-demo-projects/blob/master/2d/pong/paddle.gd`

[9] `https://github.com/rjeschke/jpa/blob/master/src/main/native/jpa.c\#L84`

occurrences of this anti-pattern extracted from *OpenResty*. We found a situation, in which a developer introduced several calls from one Lua file to the Nginx. In this example, the developer was excessively calling the function ngx.exec() from the Lua file and getting values from the configuration file. The good solution is also present in the same system. As they usually provide access as an entry point to ensure the better way of communication between Nginx and Lua[10].

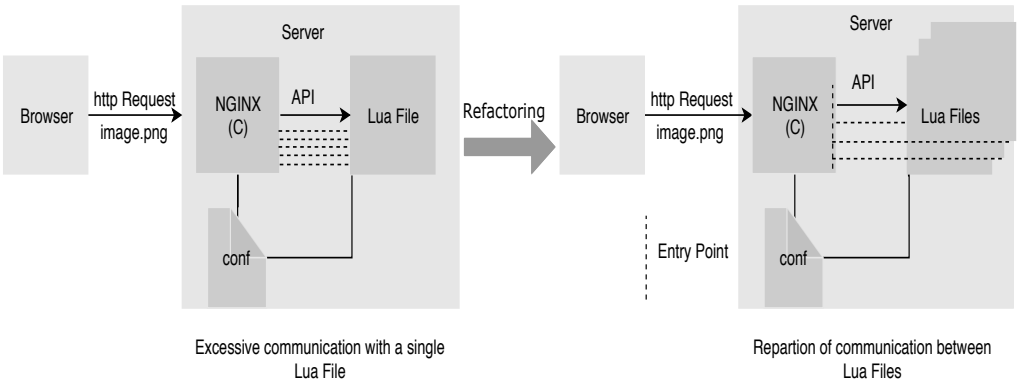

Fig. 3. Illustration Anti-Pattern - Excessive Inter-language Communication

4.2 Too Much Scattering of Multi-language Participants

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

Problem: Under time pressure developers wants to add multi-language systems code, the problem is that developers in these situations are not always sure where they should add the code. Especially that developers or maintainers do not always have a global idea about the overall architecture and design of the system. When several developers or teams are involved in the same project bugs related to changes may occur. Developers and managers would avoid these breakages in unrelated features, if the features are mixed together a change to the behavior of one may cause a bug in another feature.

Supposed Solution: Try to always separate multi-language classes to avoid breakage in unrelated features without considering the concerns. Add the foreign

[10] `https://github.com/openresty/lua-nginx-module`

code without considering the concerns and architecture of the project. Each time we estimate that the use of multi-language programming can be easier to perform a specific task, we add the foreign code without considering the classes already participating in the multi-language code and the responsibilities of each class.

Forces Toward: (1) Expose only subpart of the code and some features, and hide others features to the client; (2) Classes designed to be too simple and lightweight; (3) Adding new requirements without considering the coupling; (4) Favoring the understandability and simplicity of the classes by introducing few multi-language codes in each class; (5) Avoid breakage when applying a changes in unrelated features that are mixed together; (6) Build Large code bases over long periods of time by different people; (7) Wrong partitioning of the allocation of responsibilities of classes participating in the multi-language code.

Consequences of the Anti-Pattern: The methods and classes participating in the foreign interaction are spread through the code in a way that determining which classes are participating and which does require some effort. This code will be more difficult to maintain and refactor. It would be hard to know which classes are participating in the multi-language programming and which are not. It becomes difficult to locate and fix issues related to multi-language programming.

Forces Away: (1) Ensure a high cohesion and low coupling; (2) Merge the multi-language code in specific classes to improve the maintainability; (3) Ensure better encapsulation and Open/Closed principle; (4) Promote abstraction among classes and components.

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

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

Related Anti-Patterns: Functional Decomposition [5].

Related Patterns: Component Wrapper [17], Interface Extension [37], Facade, and Decorator [22].

Examples: This anti-pattern can be observed in a system where we have many classes participating in the multi-language programming and most of them contain only a small part involving foreign code as illustrated in figure 4. These classes are mainly mono-language but contain few foreign codes. A good solution would be to refactor the code and isolate the foreign code in a way that some classes are mainly participating in the multi-language programming and others involve only one language. We present a simple example in figure 4 to illustrate the excess of classes participating in multi-language programming. In this example, we have three classes each of them contains two native methods declaration. A good solution would be to move these methods or add a superclass if needed, that will contain all the native declaration methods, and keep these classes as inherited from this superclass. This will reduce the number of native method declaration by removing the duplicated ones. This will also reduce the scattering of multi-language participants and concerns by keeping the multi-language code concentrated only in specific classes. In the same vein, in the system *jpostal*, the classes *AddressParser* and *AddressExpander* contain each few native declaration methods that could be grouped into the same class[11]. Especially that the implementation of most of these native methods is duplicated between both of them. Other classes also from the same package contain one to two native method declaration. Another example of this anti-pattern is present in *Frostwire*. For example, the method *getWindowHandleNative()* is the only function written in C, and the window handle is used for displaying video using mplayer[12]. This method could have been grouped with other natives methods to reduce the number of classes participating in the multi-language code. There are also other ways of doing this in Java by using a video player made for Java.

4.3 Too Much Clustering of Multi-language Concerns

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

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

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

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

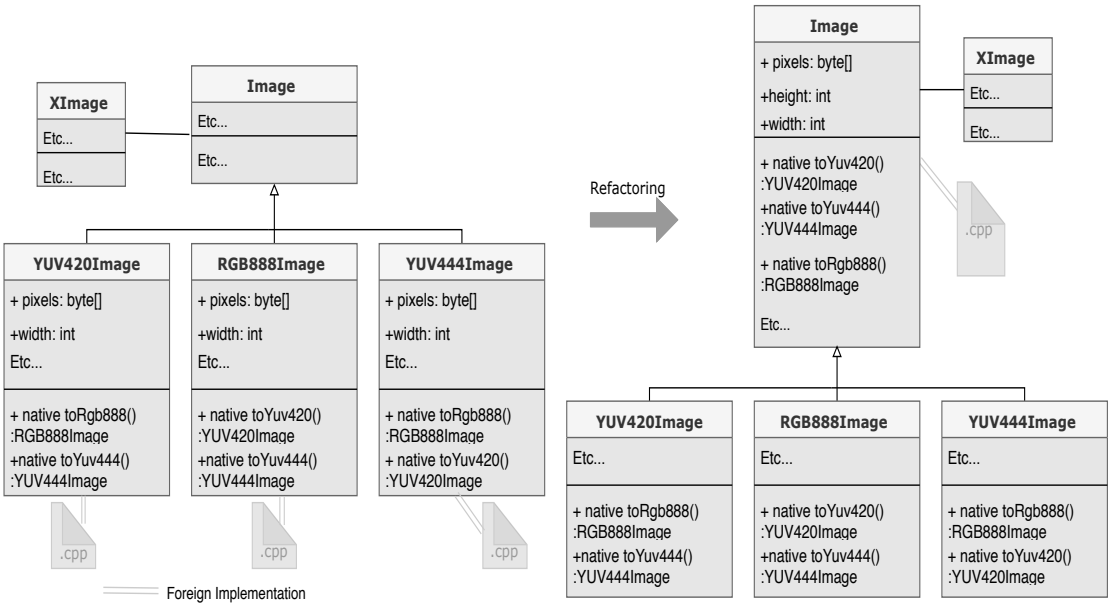

Fig. 4. Illustration Anti-Pattern - Too Much Scattering of Multi-language Concerns

Supposed Solution: The bad solution would be to always try to concentrate as much as possible the multi-language code in the same classes without considering the responsibilities related to each class. This situation can also be defined by the merge the multi-language code in a single class in a way that results in a high coupling and low cohesion. The occurrence of this anti-pattern would appear if we do not consider the concerns when adding new features or functionalities that involve the use of a multi-language code. The allocation of responsibilities between the multi-language classes is not well managed during system evolution so that one module becomes predominant regarding the other modules.

Forces Toward: (1) Inappropriate requirements allocation; (2) Class or module is given responsibilities that overlap most other parts of the project; (3) Class designed to touch multiple domains which must be decoupled from each other; (4) Iterative development where proof-of-concept code evolves over time into a prototype, and eventually, a production system evolution; (5) Classes in the project designed mainly for control or management; (6) Adding new requirements without considering the cohesion; (7) Wrong management of changes in the project by adding multi-language code to classes that are already multi-language instead of loading libraries or APIs in new classes.

Consequences of the Anti-Pattern: As a consequence of this anti-pattern, would be a negative impact on maintainability, as applying a change would require an important effort due to the complexity of understanding such code.

There is also a loose of portability and reusability as the module has more than one responsibility. If we do not consider the cohesion and concerns when adding the code, this can result in a high coupling with low cohesion.

Forces Away: (1) Depending on the number of calls decide whether components that need to talk can have direct references to each other without having to go through the manager or controller class; (2) Promote simplicity and readability of the classes; (3) High cohesion and low coupling.

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

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

Related Anti-Patterns: Too Much Scattering of Multi-language Participants, Blob, and Swiss Army Knife [5].

Related Patterns: Interface Extension [37].

Examples: This anti-pattern can be identified in multi-language systems when the multi-language code is mixed in the same classes or files without any common concerns.We believe that it is a good practice to not spread the multi-language code through the system, but this should be balanced between the context. A good solution would be to find a compromise between separating the concerns and not dispersing the multi-language code. When a change needs to be applied we should be able to easily locate the code directly associated with the change. If the concerns are well separated between the languages, it is easier for developers to work separately in different tasks or modules. Separating the concerns also help to avoid breakage in unrelated features, if the features are mixed together a change to the behavior of one may cause a bug in another feature. One of the examples, could be *React*, as it was reported to violating the separating

of concerns by mixing *JavaScript* code with *HTML*, and *CSS*[13]. We present in figure 5 an example extracted from *ZMQ JNI*[14]. In this example, native methods related to cryptographic operations are mixed in the same class as the methods used for network communication. This merging of concerns resulted in a blob multi-language class that contains 29 native declaration methods and 78 attributes.Another example, the class GodotLib which contains 25 native declaration methods[15].

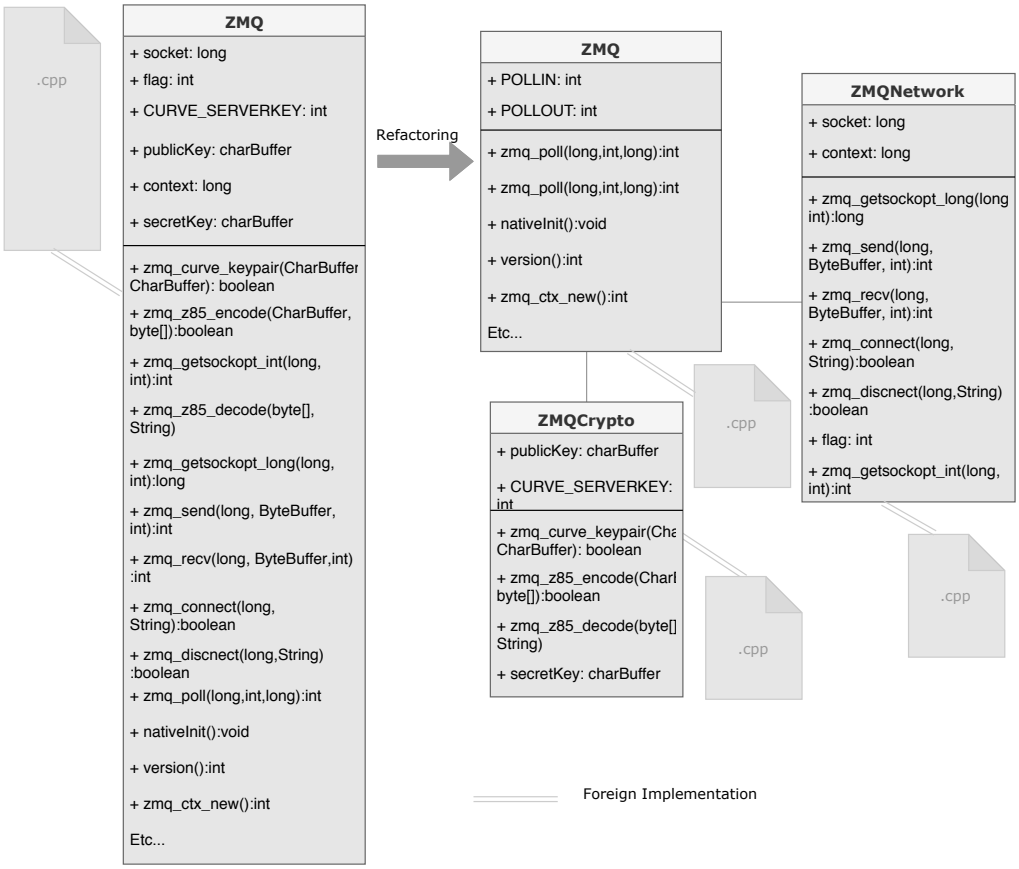

Fig. 5. Illustration Anti-Pattern - Too Much Clustering of Multi-language Concerns

[13] http://krasimirtsonev.com/blog/article/react-separation-of-concerns
[14] https://github.com/zeromq/zmq-jni/blob/master/src/main/java/org/zeromq/jni/ZMQ.java
[15] https://github.com/godotengine/godot/blob/60d910b1916305c4b0ac5f92415083995b4f7c7a/platform/android/java/src/org/godotengine/godot/GodotLib.javanativemethods

4.4 Unnecessary Use of Multi-language Programming

Context: This anti-pattern can be observed when the task can be completed in a single language in such way that we are not really taking benefit if we will introduce the usage of multi-language programming but we are adding unnecessary complexity. Excessive usage of multi-language programming may result in a loose of their benefits and adding more unnecessary complexity to the project.

Problem: This anti-pattern can result from a situation in which we are implementing a simple task or adding new features to an existing system. These features or tasks may be already available in other languages or as libraries. However, their development presents a simple task and do not require too much effort. This can also be related to the developer's experience with the programming languages. Developers have different experience and levels of interest in different programming languages. The problem is that developers do not always have a great idea about the architecture of the system to decide whether in that specific case introducing multi-language programming worth it or not.

Supposed Solution: Always favor the reuse of code. If a feature or module is already available even if in another language, then integrate the module and make your program multi-language. If we are more comfortable in a specific language that differs from the language used to implement the application, use that specific language to implement the tasks.

Forces Toward: (1) Reuse of existing resources; (2) Pressure of time delivery; (3) Management of technologies and following the trends; (4) Reuse of existing code to save the development time; (5) Take the benefit of the different programming languages; (6) Avoid reinventing the wheel; (7) Start from a working example even if implemented in another language and adapt it to the specific needs.

Consequences of the Anti-Pattern: Those kinds of systems will be difficult to maintain and understand. Especially, as reported in our fourth observation presented in section 1, systems are usually developed and maintained by different people. A maintainer may not be as comfortable with multi-language systems and may not understand why they have been used in such situations.

Forces Away: (1) Reuse components and APIs implemented in the same language as the host project; (2) Improve the reusability and portability; (3) Avoid unnecessary complexity by introducing multi-language code.

Refactoring: To refactor this anti-pattern, identify the tasks or modules that could have been written in the same language. Search for existing implementation or modules implemented in the same language that could replace the foreign code. Then, we suggest measuring the cost and impact of removing the foreign code in regards to the lifetime of the project. Then, isolate the modules and try

to migrate the features and even reproduce the bugs in the same language. We also recommend before introducing multi-language programming, to determine if we are reducing or adding more complexity. In the case where a single language can perfectly complete all the tasks, it is better to use only this language and not introduce another language. Even if at that time, a specific developer would find it easier for him to perform the tasks by reusing code written in a different language. It is always recommended to consider the maintenance cost. All the systems will be maintained and probably by another person that may not have the same preferences as the initial developer.

Benefits of the Refactoring: Avoid unnecessary complexity. This will reduce the challenges related to introducing new programming languages. It is important to avoid multi-language programming if we are losing the benefits of introducing several languages. Other benefits of applying the refactoring are to improve the understandability and readability of the code and Reduce the maintenance efforts.

Related Anti-Patterns: Overengineering.

Examples: Some occurrences of this anti-pattern have been observed in JNI systems that we analysed, in which we found simple tasks delegated to JNI code. This was also discussed in some developers' documentation[5]. Figure 6 presents a possible case of unnecessary usage of multi-language systems extracted from *JniHelpers*. In some cases, the introduction of multi-language programming presents several benefits and can be justified. For example, in the case of mathematical operations like compression or encryption, or shared library that could be better written in a language available on all platforms. In these cases, we can reduce the maintenance cost and development cost by using the existing library written in C language for example instead of re-writing the same code in several languages. However, we should always keep in mind that native code might be faster under specific circumstances. But in case of a bunch of arrays, loops, and arithmetic operations, there is no difference in performance between using java and native or a different language[16]. The solution would be when a task can be perfectly implemented in a single language always go for that language. We also found in *Telegram* occurrences of this anti-pattern. As it packages *SQLite* while there are other database types implemented in Java and recommended to be used within from Java. Shipping *SQLite* opens the application for more vulnerabilities and bugs. The same goes for shipping *FFmpeg*. It is also recommended to not mix between Media playback and security concerns. Several bugs and vulnerabilities related to *FFmpeg* have been discussed in developers' blogs and bug reports[17]. Another example found in *JniCompressions*, where native implementation where

[16] `https://www.reddit.com/r/java/comments/vr250/the_jni_is_it_worth_it/`

[17] `https://www.cvedetails.com/vulnerability-list/vendor\_id-3611/Ffmpeg.html`

used, while their functionality is already available as Apache common libraries for Java[18].

```
/* Java */
native void createJavaString();
native void nativeCreateJavaStringFromJavaString(String s);
void createJavaStringFromJavaString() throws Exception {
nativeCreateJavaStringFromJavaString(TestConstants.STRING); }
```

Fig. 6. Anti-Pattern - Unnecessary Use of Multi-language Programming

4.5 Language and Paradigms Mismatch

Context: In some cases, we can face tasks that may be better implemented in a specific language/paradigm. Also, the chosen programming language or paradigm might be inefficient for some specific tasks due to limitations of that particular language. However, the developer may be more comfortable with that specific language or paradigm.

Problem: Each programming language has its own benefits and may be more efficient for specific tasks. The choice of the programming language to use depends on how the solution is modeled and the design decision applied. Some models work better with objects, some would best be done in an iterative solution, etc. However, design decisions may change during the software development phase and the same of the programming language used in the project. These languages have different paradigms that may introduce some incompatibilities once combined. In the same vein, developers do not have the same preferences and competencies in term of programming languages. Many languages or environment decisions are made by "if you have a hammer, everything looks like a nail", developers tend to use the programming languages or tools they are familiar with.

Supposed Solution: The bad solution would be to implement the task in the language or paradigm that are easier to use but may not be the best language for that task or may introduce incompatibilities. This case may occur if we favor mono-language programming but also in the case of multi-language systems. If we do not choose the best language for the best task but always prefer to use language and paradigms with which we are more comfortable.

[18] `https://commons.apache.org/proper/commons-compress/javadocs/api-release/org/apache/commons/compress/compressors/lz4/package-summary.html`

Forces Toward: (1) Coexistence with other software; (2) Introduce benefits from low-level programming languages; (3) Ensure efficiency implementation for specific tasks; (4) Reuse of similar or same features already implemented in another language; (5) Use of available resources; (6) A prototype or a part of the code was already written in the other language and developers prefer to reuse what was already available; (7) An old project that is still used but developers avoid to apply refactoring or migrate it to new technologies.

Consequences of the Anti-Pattern: This anti-pattern can introduce problems during maintenance phases and also performance problems. As not all the languages are better used for the same tasks[19]. The same task could be written in four lines of code in Python language, however, require more than 10 lines of Java code. This may impact the understandability and maintainability of the system. Especially in the case of multi-language systems, this may cause additional overhead while debugging and maintenance of the system. Bad solutions like these contribute to the technical debt on the developers. This anti-pattern is related to our second observation presented in section 1.

Forces Away: (1) Multi-threaded safety and robustness; (2) Ensure performance and calculation time; (3) Use each language for the best purpose; (4) Ensure performance by using low-level memory for specific tasks.

Refactoring: It might be possible that a certain task can be implemented more efficiently using another comparatively lower level programming language than the primary programming language for the project. To refactor this anti-pattern we first recommend to deeply verify if the task can be isolated appropriately. If yes, then depending on the task, decide which language can be better suitable for this situation. Once the choice of the language made, search for an existing module or library implemented in that language that provides the same features. The use of another programming language or paradigm for these tasks can boost the system's overall efficiency. We suggest isolating the task to a level that any problem caused by a task can be easily traced back to the code for this task. If there is no existing library or module that can be used, the task can be programmed using the chosen programming language with proper logs and documentation that can ease the usage of the library in the system. This methodology ensures separation of concerns and availability of reusable code in different modules or even projects.

Benefits of the Refactoring: Take the benefit from each programming language and use each language for the best purpose. This can also ensure security by using programming languages that present fewer vulnerabilities. Another benefit is related to improving performance by using another lower-level language for embedded programming or OS programming.

[19] `https://stackoverflow.com/questions/1912408/appropriate\%2Dprogramming\%2Dlanguages\%2Dfor\%2Ddifferent\%2Dproblems`

Related Anti-Patterns: Blob [5].

Related Patterns: Object System Layer [29], Wrapper, and Facade [22].

Examples: One of the observed example of occurrences of this anti-patterns was while sending files in Python. The system *python-telegram-bot* also contains occurrences of this anti-pattern. Several issues have been reported when sending files in Python[20]. In the case where packets are checked for an acknowledgment then the transmission speed in Python is much lower than that of C programming language and may lead to timeout issues. File transmission is a task that can be easily isolated, and therefore programmed in C language, which can be converted into a dynamic library for use in Python. In this kind of situation, it would be better to isolate the task and provide an external library. We will benefit from the advantage that is introduced by the different programming language and we use the right language for the right task. We present in figure 7 another example of occurrences of this code smell extracted from *jMonkeyEngine*[21]. In this example, JMonkeyEngine uses Java to process a lot of mathematical operations mostly related to terrain generation using the method *load()*. This could have been offloaded to C(++) and ensure better performance for each device as the system already involve the C(++) language.

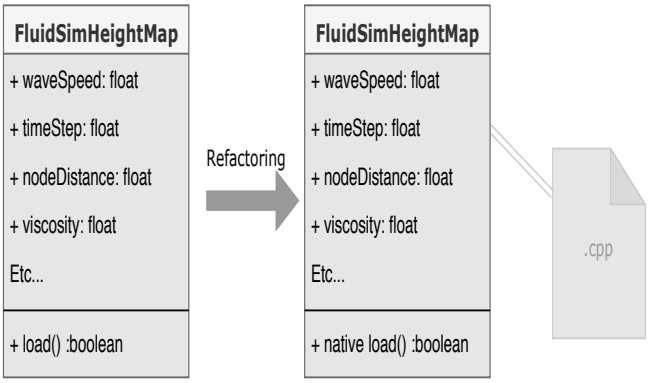

Fig. 7. Illustration Anti-Pattern - Language and Paradigms Mismatch

4.6 Project Migration Language Related Issues

Context: Developers and companies frequently face situations where projects fail or introduce several issues to be migrated. This can also be faced when

[20] https://github.com/python-telegram-bot/python-telegram-bot/issues/533

[21] https://github.com/jMonkeyEngine/jmonkeyengine/blob/master/jme3-terrain/src/main/java/com/jme3/terrain/heightmap/FluidSimHeightMap.java

modernising application from old technologies to the new trends and advantages available in the market. Another case is, where applications or websites were designed as a prototype or for internal usage. But then started to be used by an important number of users. For this kind of reasons companies often migrate their applications. Another illustration of this anti-pattern is that there are some utility tools that are not updated to support the latest and advanced features of new technologies. This can cause restrictions on advancements and updates to the project if these tools are not replaced. Some other tools may be migrated from one language to another from one technology to another. These systems, often become multi-language systems as a subset of the system remain in the old language and new features should be implemented in another language.

Problem: Usually systems are implemented under time delivery pressure or are designed as a prototype for internal usage. These systems are usually not implemented in a way that easily allows future migration and compatibility with new technologies. New programming languages and technologies also appear every day and with time, these technologies often become obsolete. When migrating project it is also challenging to migrate business rules. In some cases, the programming language or technologies used in the past, may not be still used by an important number of developers. Martin Fowler discussed this common problem as it is really more complex to migrate systems than what it seems[22]. He explained that even when adding new features, old stuff has to remain, including old bugs that need to be added to the migrated version of the system. He introduced the concept of a strangler application pattern as a way of handling the release of the refactored code in a large application. We highly recommended when developing a new application to make it easier to be strangled in the future. Several studies also in the literature discussed the common issues and challenges related to the migration of such application [38].

Supposed Solution: If the tool supports external libraries, then dynamic libraries are created focusing on fulfilling the requirements at hand. If the tools have no support for external libraries, new tools are designed for that specific requirement or additional third-party software are used.

Forces Toward: (1) Legacy Configurations; (2) Business pressure; (3) Prioritising the delivery of a working version and do not consider the maintenance activities and evolution after delivery; (4) The project designed as a prototype or one time project not designated to add new features or be migrated with new technologies; (5) Not considering the extensibility, only the delivery process in that present time; (6) Design not oriented to support important changes and allowing evolvability and openness; (7) Project based entirely upon marketing and industry need, and not consider future needs; (8) Lack of process management; (9) Companies looking for a quick and cheap transition to a client/server architecture.

[22] `https://www.martinfowler.com/bliki/StranglerApplication.html`

Consequences of the Anti-Pattern: In long term projects, generally some utility/third-party tools are developed and used to interact with the primary system. These tools are developed with a specific aim in mind, and their design might not have been given enough attention to supporting extensions according to the latest technology trends and new programming languages. As new concepts are being implemented in the form of packages and libraries constantly, if the project dependency on these obsolete tools is high, then the entire project can become obsolete. The libraries might be developed to solve specific problems at hand, but the lack of updates and bad design for a tool in most cases causes additional problems with time. Moreover, in most cases, whenever a third party tool or library is used, only a small subset of its overall features is used in the project, which results in additional technical overhead for the people working with them. Migration issues are a common discussion between developers. Especially when migration a project from one language to another (e.g. from COBOL to Java), developers are usually asking for any learned lessons or practices to avoid common migration issues[23].

Forces Away: (1) Adapt to a changing world and technologies; (2) Coexistence with other software and technologies; (3) Allow extensibility and reusability; (4) Incremental design process; (5) Choose language with active and important community; (5) Preserve several years of development, while greatly enhancing performance and flexibility.

Refactoring: To refactor this anti-pattern, we recommend first to understand the whole architecture of the system to be migrated. Then, chose the languages and technologies that will be used for the new version. Depending on the languages and technologies, they may be some existing tools that can help during the migration phase. It is also important, to consider making the system more flexible to future migrations. Martin Fowler introduced a possible solution to consider a strangler application over a cut-over rewrite. He also suggested as good practice when designing a new application to make it easier to be strangled in the future. A good solution would be to always keep a future vision when implementing a system. New technologies and languages appear every day. Each of them introduces new advantages and may solve specific challenges. The systems should be designed in a way to allow extensions, especially for multi-language systems usage. This will allow for smooth addition of future modifications and new features. Developers should ensure that tool support is always as good as expected.

Benefits of the Refactoring: Consider future extensions and reduce the costs and risks of project migration. This also allows the project to stay in the market and easily migrated to new technologies. Another benefit is related to the post-delivery as it ensures a better lifetime of the project once delivered.

[23] `https://stackoverflow.com/questions/1029974/experience-migrating-legacy-cobol-pl1-to-java`

Related Anti-Patterns: Continuous Obsolescence and Autogenerated Stovepipe [5].

Related Patterns: Strangler Application [39].

Examples: An example of this anti-pattern would exist in each application that failed or introduced high cost, to be migrated from one technology or language to another. Martin Fowler also discussed examples of this anti-pattern. One example of this anti-pattern could be faced by a company with COBOL systems that cannot be migrated to new hardware for lack of appropriate compilers. Developers would have to deal with different tools and languages due to this migration issue[24]. Some of these issues were reported in one of our current studies in which we surveyed developers about the challenges of multi-language systems. The legacy tools should be replaced with the latest feature-rich technologies that will provide more area for improvement and innovation. It would always be better to develop and maintain one utility tool with new technology than to maintain multiple legacy tools using a different set of technologies. Another example of this anti-pattern is the features that are available in one language but in the other language. As an example we have the code assistance in Java but not in C language. In the literature, we also found occurrences of this anti-pattern presented as an industrial report when migrating an airport management system from a Bull mainframe using COBOL programming language and IDS as a database to a distributed UNIX platform using Java and Oracle [38]. They presented the challenges and issues related to such migration. Previously, two attempts have already been made to migration this application from COBOL to Java but both of them failed. They also argued that it is much more difficult to migrate an existing application than to develop a new one starting from scratch. As in commercial applications, users are expecting to have all of the old features plus new ones. Another example of this situation was the case for *Microsoft*, when they rewrote their compiler[25]. The same for *Facebook*, with the increase of its popularity, PHP could not support the volumes they process. For that, they migrated the PHP into C++ thence machine code[26].

5 Code Smells for Multi-language Systems

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

[24] https://stackoverflow.com/questions/1029974/experience-migrating-legacy-cobol-pl1-to-java
[25] https://medium.com/microsoft-open-source-stories/how-microsoft-rewrote-its-c-compiler-in-c-and-made-it-open-source-4ebed5646f98
[26] https://softwareengineering.stackexchange.com/questions/176435/why-does-facebook-convert-php-code-to-c

5.1 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.

Supposed Solution: We usually have to decide whether we pass an object that has multiple fields or passes 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: his 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*[4]. 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 8

presents an example of occurrences of this code smell. While figure 9 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.

```
/* 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;}
```

Fig. 8. Code Smell - Passing Excessive Objects

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

Fig. 9. Refactoring - Passing Excessive Objects

5.2 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.

Supposed 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.

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 10 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;}
```

Fig. 10. Code Smell - Unnecessary Parameters

5.3 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.

Supposed 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.

5.4 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.

Supposed 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.

5.5 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.

Supposed 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. This code smell is related to the third observation presented in section 1 related to the correctness.

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: he 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[4] [27]. Most of the systems that we analysed were not always implementing a proper way to handle the exception as shown in figure 11, 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 12 presents a refactoring example extracted from *IBM Developer Site*[4]. 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.

5.6 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.

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

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

Fig. 11. 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);
```

Fig. 12. Refactoring - Not Handling Exceptions Across Languages

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.

Supposed 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. This code smell is related to the third observation presented in section 1 related to the correctness.

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[28]. 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 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 13 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 14 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");
```

Fig. 13. Code Smell - Assuming Safe Multi-language Return Values

5.7 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.

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

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

Fig. 14. Refactoring - Assuming Safe Multi-language Return Values

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.

Supposed 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: he 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 15[4]. 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 our first observation discussed in the introduction, related to the performance. As presented in the example[4], using caching field IDs will take 3,572 ms to run 10,000,000 times 15. However, without using the cache as illustrated in 16, it takes 86,217 ms. Using this code smell the task takes 24 times longer than without the occurrences of this code smell.

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

Fig. 15. 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;
```

Fig. 16. Refactoring - Not Caching Objects' Elements

5.8 Not Securing Libraries

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

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

Supposed 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 17. Depending on the languages, we recommend to always secure the loading library by using available methods for the specific language.

5.9 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.

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

Fig. 17. Securing Library Loading

Problem: Project was designed as a prototype and do not consider future extensions and adaption to new platforms.

Supposed 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 18. 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 19. 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) {...}
```

Fig. 18. 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");
```

Fig. 19. Refactoring - Hard Coding Libraries

5.10 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.

Supposed 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 20 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"); }
```

Fig. 20. Refactoring - Not Using Relative Path to Load the Library

5.11 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.

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 21 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; }
```

Fig. 21. Refactoring - Memory Management Mismatch

5.12 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.

Supposed 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 22 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;}
```

Fig. 22. Code smell - Local References Abuse

6 Threats to Validity

We now discuss threats to validity of our methodology and the reported anti-patterns and code smells.

Threats to internal validity: We used the well-know, open-source repositories *GitHub* and *OpenHub* to identify and obtain multi-language systems. We also used well-known developers' documentations, bug reports, and developers' blogs, such as *StackOverflow*, *IBM Developers*, *developer.android*, and *Bugzilla* to extract practices. Hence, we limited threats to the internal validity, although we did not identify exhaustively all existing anti-patterns/code smells. Moreover, we followed a systematic method to identify and report multi-language anti-patterns and code smells.

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

Threats to reliability validity: We attempted to provide all the necessary information needed to reproduce our study here and online[29], including our developers' survey. Hence, we believe to have minimised threats to its reliability.

7 Conclusions and Future Work

Most of the existing systems are multi-language systems and consist of components written in several, different programming languages. Multi-language systems provide many benefits because developers can reuse existing code and take

[29] `http://www.ptidej.net/downloads/replications/europlop19/`

advantage of existing libraries, even if written in different programming languages [15]. Multi-language systems also raised with the need to include and accommodate legacy code. However, multi-language systems also present challenges to developers: they are difficult to develop, maintain, and evolve because they are more complex than mono-language systems.

To the best of our knowledge, good and bad practices in the development, maintenance, and evolution of multi-language systems are scattered across various resources, including few academic papers, some blogs, programming-language specifications, etc.

Therefore, in this paper, we present the steps followed for studying these resources and report on 12 code smells and six anti-patterns that we borrowed, observed, and–or inferred from these resources. These practices should help developers and researchers to handle the complexity of multi-language systems. We followed and adapted the template provided by *WikiWikiWeb*.

In future work we will (1) survey developers about these anti-patterns and code smells, (2) combine multi-language design patterns and anti-patterns to relate them with one another, (3) create a pattern language that could relate multi-language design patterns, design anti-patterns, idioms, and code smells with one another, (4) investigate their impact on quality attributes, and (5) implement tools to identify and correct their occurrences.

References

1. P. L. Roden, S. Virani, L. H. Etzkorn, and S. Messimer, "An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes," in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on.* IEEE, 2007, pp. 171–179.
2. D. Galin, *Software quality assurance: from theory to implementation.* Pearson Education India, 2004.
3. E. Shihab, "Practical software quality prediction," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on.* IEEE, 2014, pp. 639–644.
4. C. Alexander, S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, and I. Fiksdahl-King, *A pattern language.* Gustavo Gili, 1977.
5. 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.
6. M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.
7. C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?" *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213–1231, 2012.
8. F. Khomh and Y.-G. Gueheneuce, "Do design patterns impact software quality positively?" in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on.* IEEE, 2008, pp. 274–278.

9. F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.
10. 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.
11. 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.
12. R.-H. Pfeiffer and A. Wasowski, "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.
13. 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.
14. ——, "Multilingual source code analysis: State of the art and challenges," in *Open Source Systems & Technologies (ICOSST), 2015 International Conference on*. IEEE, 2015, pp. 170–175.
15. 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.
16. 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.
17. 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.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
19. A. Malinova, "Design approaches to wrapping native legacy codes," *Scientific works, Plovdiv University*, vol. 36, pp. 89–100, 2008.
20. 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.
21. M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 62–72.
22. 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.
23. R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
24. B. F. Webster, *Pitfalls of object oriented development*. Book, 1995.
25. Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 393–402.

26. A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 242–251.
27. 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.
28. D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 437–446.
29. M. Goedicke, G. Neumann, and U. Zdun, "Object system layer," *5th European Conference on Pattern Languages of Programms (EuroPLoP '2000)*, 2000.
30. ——, "Message redirector," *6th European Conference on Pattern Languages of Programms (EuroPLoP '2001)*, 2001.
31. 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.
32. A. Osmani, *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. " O'Reilly Media, Inc.", 2012.
33. 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.
34. 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.
35. 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.
36. S. Liang, *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.
37. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.
38. H. M. Sneed, "Migrating from cobol to java," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–7.
39. M. Fowler, "Strangler application," 2004. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html