CHAPTER 8 A DETECTION APPROACH FOR MULTI-LANGUAGE DESIGN SMELLS

8.1 Chapter Overview

In the previous chapter, we defined a set of 15 multi-language design smells that we cataloged in order to help improve the quality of multi-language systems. Therefore, in this chapter, we propose an approach for the detection of the design smells introduced in Chapter 7. In this chapter, we start by presenting the implementation details of our approach. Then we present the rules used to define and detect the multi-language design smells. We also present the results of the evaluation of the proposed approach.

8.2 Approach Definition

Because no tools are available to detect design smells in multi-language systems, we build a new detection approach named `MLSInspect`. We used srcML[1], a parsing tool that converts source code into srcML, which is an XML format representation. The srcML representation of the source code adds syntactic information as XML elements into the source code text. Listing 8.2 presents the srcML representation of the code snippet presented in Listing 8.1. The main advantage of srcML, is that it supports different programming languages, and generates a single XML file combining source code files written in more than one programming language. Languages supported in the current version of srcML include Java, C, C++, and C#[2]. However, this could be extended to include other programming languages [125]. SrcML provides a wide variety of predefined functions that could be easily used through the XPath to implement specific tasks. XPath is frequently used to navigate through XML nodes, elements, and attributes. In our case, it is used to navigate through srcML elements generated as an XML representation of a given project. It allows access to all levels of information in a source-code document, *i.e.,* lexical, structural, syntactic and documentary. The ability to address source code using XPath has been applied to several applications [126].

Part of the content of this chapter is published in: Mouna Abidi, Md Saidur Rahman, Moses Openja, Foutse Khomh. "Are Multi-language Design Smells Fault-prone? An Empirical Study", *Published in Transactions on Software Engineering and Methodology (TOSEM), 2020, ACM.*

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

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

Listing 8.1 Example of Java Code

```
public class HelloWorld {

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

Listing 8.2 Example of Java Code Converted to SrcML

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

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

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

Our detection approach reports smell detection results for a given system in a CSV file. The report provides detailed information for each smells detected such as smell type, file location, class name, method name, parameters (if applicable). The approach also allows to post-process the results and create a summary file. The summary results provide a CSV file that details for each specific file or class the total number of occurrences of each type of smell, the date and other information related to that specific version of the system.

8.3 Detection Rules

The detection approach is based on a set of rules defined from the documentation of the design smells presented in Chapter 7. Those rules were validated by the pattern community during

the Writers' workshop organised in the context of the pattern conference *i.e.,* (EuroPlop) to document and validate the smells. For example, for the design smell *Local Reference Abuse*, we considered cases where more than 16 references are created but not deleted with the *DeleteLocalRef* function. The threshold 16 was extracted from developers blogs discussing best practices and the Java Native Interface specification [41][3, 4]. We present in the following the smell detection rules of the proposed approach. These rules are applied on the srcML elements generated as an XML representation of a given project. Since the smells described in this thesis are multi-language smells, the following rules detect the occurrences of smells by using the XPath queries in the srcML representation of the source code that contains Java and C/C++ native code.

1. **Rule 1: *Not Handling Exceptions***

$$(f(y) \mid f \in \{GetObjectClass, FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID\})$$
$$\textbf{\textit{AND}}\ (isExceptionChecked(f(y)) = False\ \textbf{\textit{OR}}\ ExceptionBlock(f(y)) = False)$$

   Our detection rule for the smell *Not Handling Exceptions* is based on the existence of call to specific JNI methods requiring an explicit management of the exception flow. The JNI methods (*e.g.,* FindClass) listed in the rule should have a control flow verification. The parameter y presents the Java object/class that is passed through a native call for a purpose of usage by the C/C++ side. Here, *isExceptionChecked* allows to verify that there is an error condition verification for those specific JNI methods, while *ExceptionBlock* checks if there is an exception block implemented. This could be implemented using Throw() or ThrowNew() or a return statement that exists in the method in case of errors.

2. **Rule 2: *Assuming Safe Return Value***

$$x := f(y) \mid f \in \{FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID\}\ \textbf{\textit{AND}}\ isErrrorChecked(x) = False\ \textbf{\textit{AND}}\ IsReturn(x) = True$$

   This rule is quite similar to the previous rule. However, it considers the return value from the native code. Indeed, the JNI methods called in this context are used for

[3] https://www.cnblogs.com/cbscan/articles/4733508.html

[4] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#global_local

specific calculation and the result then needs to be passed as a method return value to the Java side. Here, x presents the native variable used within the method to receive the returned value and perform computation on the Java side. *isErrrorChecked(x)* allows to verify if there is an error condition verification applied to the variable x that will be returned back to the Java code (*IsReturn(x)=True*). The use of the variable x as a return value by a native method without any check of its correctness will introduce smell of type *Assuming Safe Return Value* given other conditions hold.

3. **Rule 3: *Not Securing Libraries***

$$IsNative(Lib) = True \ \textbf{AND} \ loadedWithinAccessBlock\ (Lib) = False$$

This rule implies that in the Java code, a native library is used (*IsNative(Lib) = True*) and that this library is loaded outside a block *AccessController.doPrivileged* without a try and catch statements for safe handling of potential exceptions. This introduces smell of type *Not Securing Libraries*.

4. **Rule 4: *Hard Coding Libraries***

$$IsNative(Lib) = True \ \textbf{AND} \ AccessiblePath(Lib) = False \ \textbf{AND} \ OsBlock(m) = True$$

This rule implies that in the Java code, a native library (*Lib*) is used in a native method m and that the path used for accessing that library is an absolute path while the code loading the library depends on the operating systems. Here, the access to libraries is hard coded for specific operating system rather than implementing a platform independent access mechanism for libraries. This limits the portability of the code and may cause issues in accessing the libraries for different operating systems.

5. **Rule 5: *Not Using Relative Path***

$$IsNative(Lib) = True \ \textbf{AND} \ RelativePath(Lib) = False$$

This rule implies that in the Java code, a native library is used. However, the native library loaded from an absolute path and not from a relative path.

6. **Rule 6: *Too Much Clustering***

   NbNativeMethods (C) >= MaxMethodsThreshold ***AND*** *IsCalledOutside(m) = True*

   This rule detects cases where the total number of native methods (*NbNativeMethods*) within any class C is equal to or higher than a specific threshold while those methods m are used by other classes and not only the one where they are declared (*IsCalled-Outside(m) = True*). In our case, we used the default values for the threshold eight. However, all the thresholds could be easily adjusted.

7. **Rule 7: *Too Much Scattering***

   $$NBNativeClass(P) >= MaxClassThreshold$$
   $$\textbf{AND } (NbNativeMethods(C) < MaxMethodsThreshold \textbf{ AND } C \in P)$$

   The smell of type *Too Much Scattering* occurs when the total number of native classes in any package P (*NBNativeClass(P)*) is more than a specific threshold (*MaxClassThreshold*) for the number of maximum native classes. In addition, each of those native classes C contains a total number of native methods (*NbNativeMethods(C)*) less than a specific threshold (*MaxMethodsThreshold*) *i.e.,* the class does not contain any smell of type *Too Much Clustering*. We used default values for the threshold three for the minimum number of classes with each a maximum of three native method each.

8. **Rule 8: *Excessive Inter-language Communication***

   (NBNativeCalls(C,m) > MaxNbNativeCallsThreshold) ***OR***
   (NbNativeCalls(m(p)) > MaxNativeCallsParametersThreshold) ***OR***
   ((NBNativeCalls (m) > MaxNbNativeCallsMethodsThreshold) ***AND*** *IsCalledInLoop(m) = True)*

   The smell *Excessive Inter-language Communication* is detected based on the existence of at least one of these three scenarios. First, in any class C the total number of calls to a particular native method m exceeds the specified threshold (*NBNativeCalls(C,m) > MaxNbNativeCallsThreshold*). Second, the total number of calls to the native methods m with the same parameter p exceeds the specific threshold (*MaxNativeCallsParame-*

tersThreshold). Third, the total number of calls to a native method m within a loop is more than the defined threshold (*(MaxNbNativeCallsMethodsThreshold*).

9. **Rule 9: *Local References Abuse***

$$(NbLocalReference(f_1(y)) > MaxLocalReferenceThreshold) \textbf{ AND}$$
$$(f_1(y) \mid f_1 \in \{GetObjectArrayElement, GetObjectArrayElement, NewLocalRef,$$
$$AllocObject, NewObject, NewObjectA, NewObjectV, NewDirectByteBuffer,$$
$$ToReflectedMethod, ToReflectedField\}) \textbf{ AND}$$
$$(\nexists\ f_2(y) \mid f_2 \in \{DeleteLocalRef, EnsureLocalCapacity\})$$

The smell *Local References Abuse* is introduced when the total number of local references (*NbLocalReference(*f_1*(y))*) created inside a called method exceeds the defined threshold and without any call to method `DeleteLocalRef` to free the local references or a call to method `EnsureLocalCapacity` to inform the JVM that a larger number of local references is needed.

10. **Rule 10: *Memory Management Mismatch***

$$(mem \leftarrow f_1(y) \mid f_1 \in \{GetStringChars, GetStringUTFChars,$$
$$GetBooleanArrayElements, GetByteArrayElements,$$
$$GetCharArrayElements, GetShortArrayElements,$$
$$GetIntArrayElements, GetLongArrayElements,$$
$$GetFloatArrayElements, GetDoubleArrayElements, GetPrimitiveArrayCritical,$$
$$GetStringCritical\})$$
$$\textbf{AND}\ \ (\nexists\ f_2(mem) \mid f_2 \in \{ReleaseGetStringChars, ReleaseGetStringUTFChars,$$
$$ReleaseGetBooleanArrayElements, ReleaseGetByteArrayElements,$$
$$ReleaseGetCharArrayElements, ReleaseGetShortArrayElements,$$
$$ReleaseGetIntArrayElements, ReleaseGetLongArrayElements,$$
$$ReleaseGetFloatArrayElements, ReleaseGetDoubleArrayElements,$$
$$ReleaseGetPrimitiveArrayCritical, ReleaseGetStringCritical\})$$

As discussed earlier, JNI offers predefined methods to manage the access of reference types that are converted to pointers. These methods are used to create pointers and to

allocate the corresponding memory. The rule described here allows to detect the native implementation in which the memory was allocated by calling one of these allocation methods, however, the memory allocated was never released. The rule detects situations in which 'get' methods are used to allocate memory for specific JNI elements that are not released after usage by calling the corresponding 'release' methods.

11. **Rule 11: *Not Caching Objects***

$$
\begin{aligned}
&((Parameter(m,p) = Object)\ \textbf{AND}\\
&\quad((NbCalls(C,m) >= MaxNbCallsThreshold)\ \textbf{OR}\ (IsLoop(m) = True\\
&\qquad\textbf{AND}\ NoOfIterations >= MaxCountThreshold))\\
&\qquad\textbf{AND}\ (IsCalled(m, f_n(y)) = True)\\
&\quad\textbf{AND}\ (f_n(y) | f_n \in \{GetFieldID, GetMethodID, GetStaticMethodID\}))\\
&\textbf{OR}\ ((Parameter(m,p) = Object)\ \textbf{AND}\ (IsCalledInMethod(m, f_n) = True\\
&\qquad\textbf{AND}\ NbCalls(f_n(y)) >= MaxNbCallsThreshold)\ \textbf{AND}\\
&\qquad(f_n(y) | f_n \in \{GetFieldID, GetMethodID, GetStaticMethodID\}))
\end{aligned}
$$

This rule allows to detect occurrences of the smell *Not Caching Objects* based on two situations. The first one is where the total number in which ids related to the same object p are looked up for the same class C through JNI allocation methods is greater than or equal to a specific threshold or the method is called within a loop. Indeed, the ids returned for a given class C remain the same for the lifetime of the JVM execution. Considering that we have a native method m and one of its parameter p is a Java object (*Parameter(m,p)=Object*), this type is considered in the native code as a reference type. Thus, unlike primitive types, its element could not be accessed directly by the native code but should be accessed through the usage of the methods defined in (*IsCalled(m,*$f_n(y)$) = True). In this first scenario, the total number of calls from the Java code to a native method m that is defined in a class C exceeds a specific threshold (*i.e., NbCalls(C,m) >= MaxNbCallsThreshold*) or the method is called within a loop. In the second scenario, the number of times the same id for an object p is looked up inside the same method m (*IsCalledInMethod(m,* f_n*)=True*) more than a given threshold even if the method m is called only once (*NbCalls(*$f_n(y)$*) >= MaxNbCallsThreshold*). This last scenario includes the total number of calls to the predefined methods (*NbCalls(*$f_n(y)$) independent of the total number of calls to the method itself.

12. **Rule 12: *Excessive Objects***

$$(Parameter(m,p) = Object) \textbf{ AND } (IsCalledInMethod(m, f_1) = True) \textbf{ AND}$$
$$(NbCalls(f_1(y)) >= MaxNbCallsThreshold) \textbf{ AND}$$
$$(f_1(y) | f_1 \epsilon \{GetObjectField, GetBooleanField, GetByteField, GetCharField,$$
$$GetShortField, GetIntField, GetLongField, GetFloatField, GetStaticObjectField\})$$
$$\textbf{AND } (\nexists\, f_2(y) \mid f_2 \in \{SetObjectField, SetBooleanField, SetByteField, SetCharField,$$
$$SetShortField, SetIntField, SetLongField, SetFloatField, SetStaticObjectField\})$$

This rule identifies situations in which a JNI object is passed as a parameter (*Parameter(m,p)=Object*) to the native code. In this context the total number of calls to allocation methods to retrieve its field id in the same method is higher than a specific threshold (*i.e., NbCalls($f_1(y)$) >= MaxNbCallsThreshold*), without a call to corresponding set functions to set the object fields by the native code. However, as described in Chapter 7, having the total number of calls to allocation methods higher than the threshold is not considered as a smell only in situations where the purpose of those calls was to set the object fields by the native code.

13. **Rule 13: *Unused Method Implementation***

IsNative(m) = True ***AND*** *IsDeclared(m) = True* ***AND*** *IsImplemented(m) = True* ***AND*** *IsCalled(m)=False*

This rule allows to capture the native functions m (*IsNative(m) = True*) implemented in the C/C++ (*IsImplemented(m) = True*), declared in Java with the keyword *native* but never used in the Java code (*IsCalled(m)=False*). It looks for the native methods that are declared using the keyword *native* with a header in the Java code and looks for the corresponding native implementation nomenclature.

14. **Rule 14: *Unused Method Declaration*:**

IsNative(m)=True ***AND*** *IsDeclared(m)=True* ***AND*** *IsImplemented(m)=False*

Native functions declared in Java with the keyword *native* (*IsDeclared(m) = True*) that are not implemented in C/C++ (*IsImplemented(m)=False*). This rule allows to retrieve the native methods that are declared with a header in the Java code using the keyword *native* and checks for the corresponding implementation nomenclature. However, those methods were never used or even implemented in the C/C++ code.

15. **Rule 15: *Unused Parameters***

$$(IsNative(m(p)) = True \ \textbf{\textit{AND}} \ IsDeclared(m(p)) = True$$
$$\textbf{\textit{AND}} \ IsImplemented(m(p)) = True \ \textbf{\textit{AND}} \ IsParameterUsed(p) = False$$

This rule reports the method parameters that are used in the Java native method declaration header using the keyword *native* (*IsDeclared(m(p))=True*). However the parameter is never used in the body of the implementation of the methods, apart from the first two arguments of JNI functions in C/C++. The rule checks if the parameter p is used in the corresponding native implementation (*IsParameterUsed(p) = False*).

8.4 Evaluation of the Detection Approach

In order to evaluate the detection approach, we started by selecting the object systems that will be used for the evaluation. We considered six open source projects that we analyzed in a prior study and that were part of the definition of the design smells as presented in Chapter 7 [34, 49, 120]. Those systems are open source projects hosted in GitHub, they are highly active, and have the characteristic of being developed with Java and C/C++. Another selection criteria was that those systems have different size and belong to different domains. Table 8.1 presents the studied systems. Similar to previous studies evaluating detection approaches for design patterns and design smells [78, 90, 94, 95, 127], we decided to measure the recall and precision of the detection approach as they they are the adequate performance metrics for such evaluation. To assess the recall and precision of our detection approach, we evaluated the results of the approach at the first level by creating dedicated unit tests for the detector of each type of smell to confirm that the approach is detecting the smells introduced in our pilot project. The pilot project was the project we developed with the occurrences of the smells along with the clean code without any smell to test and validate our approach. This explains the 100% precision and 100% recall for all the smells. We relied on six open source projects used in previous works [34, 49] on multi-language design smells. For each of the systems, we manually identified occurrences of the studied design smells. Two of the research team members independently identified occurrences of the design smells in JNI open source projects, and resolved disagreements through discussions with the whole research team. Using the ground truth based on the definition of the smell and the detection results, we computed *precision* and *recall* as presented in Table 8.1 to evaluate our smell detection approach. Precision computes the number of true smells contained in the results

of the detection tool, while recall computes the fraction of true smells that are successfully retrieved by the tool. From the six studied systems, we obtained a precision between 88% and 99%, and a recall between 74% and 90%. We calculate precision and recall based on the following Equations (8.1) and (8.2) respectively:

$$Precision = \frac{\{existing\ true\ smells\} \bigcap \{detected\ smells\}}{\{detected\ smells\}} \tag{8.1}$$

$$Recall = \frac{\{existing\ true\ smells\} \bigcap \{detected\ smells\}}{\{existing\ true\ smells\}} \tag{8.2}$$

We present in Table 8.2 the results of the evaluation of the performance of our design smell detection approach. As explained earlier, for the pilot project, we have 100% precision and 100% recall for all the smells. For other projects, the precision was evaluated by a manual inspection of all the identified occurrences of multi-language design smells by the approach. The recall was evaluated by a manual investigation of the Java and native code (C/C++) to capture occurrences may have not been detected by the approach. The reasons for false positives (FP) and false negatives (FN) are mainly related to the alternative implementations of the multi-language code that do not follow JNI specification guidelines and therefore are not currently covered by our approach. Indeed, our approach considers the JNI implementation with the appropriate naming convention as described in the JNI specification (*e.g.,* using the *native* keyword in the Java native method declaration, using JNIENV, JNIEXPORT, JNICALL, and Java_ClassName_methodname) [128]. Our approach may present some limitations for the smell *Local References Abuse* in situations in which some specific methods are used to ensure the memory capacity. However, as per our manual analysis when defining the smells, those methods are not considered relevant to detect the smell and are not frequently used. We are aware that in similar situation, the approach may result in false positives. For the smells *Assuming Safe Return Value* and *Not Handling Exceptions*, the false negatives in *Conscrypt* were related to an intermediate step that made the detection harder. In this step, the native value was checked before returning it to the Java code.

Table 8.1 Validation of the Smell Detection Approach

Systems	True Positive	False Positive	False Negative	Recall	Precision
openj9	3293	137	250	93%	96%
rocksdb	922	50	136	87%	95%
conscrypt	556	29	133	80%	95%
pilot project	32	0	0	100%	100%
pljava	511	5	53	90%	99%
jna	375	50	127	74%	88%
jmonkey	2210	142	185	92%	94%

Table 8.2 Validation Results for Each Type of Smells

Project	**Evaluation**	EIC	TMC	TMS	UMD	UMI	UP	ASRV	EO	NHE	NCO	NSL	HCL	NURP	MMM	LRA
PilotProject	FP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Precision	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	FN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
conscrypt	FP	0	0	-	6	-	15	3	-	5	-	0	-	0	0	0
	Precision	100%	100%	-	98%	-	95%	0%	-	0%	-	100%	-	100%	100%	100%
	FN	0	0	-	0	-	132	0	-	0	-	0	-	0	0	1
	Recall	100%	100%	-	100%	-	67%	100%	-	100%	-	100%	-	100%	100%	80%
pljava	FP	4	0	0	1	0	0	-	-	0	-	-	-	-	0	-
	Precision	95%	100%	100%	98%	100%	100%	-	-	100%	-	-	-	-	100%	-
	FN	3	0	0	42	0	1	-	-	0	-	-	-	-	8	-
	Recall	96%	100%	100%	67%	100%	99%	-	-	100%	-	-	-	-	50%	-
openj9	FP	9	0	0	29	-	95	0	-	3	-	0	-	0	1	0
	Precision	96%	100%	100%	95%	-	95%	100%	-	98%	-	100%	-	100%	94%	100%
	FN	37	0	0	40	-	76	4	-	86	-	0	-	0	4	1
	Recall	85%	100%	100%	94%	-	96%	66%	-	63%	-	100%	-	100%	81%	80%
rocksdb	FP	24	2	5	2	-	17	0	-	0	-	0	0	0	0	-
	Precision	96%	96%	92%	88%	-	93%	100%	-	100%	-	100%	100%	100%	100%	-
	FN	91	4	5	0	-	31	0	-	0	-	3	0	0	2	-
	Recall	86%	92%	92%	100%	-	88%	100%	-	100%	-	73%	100%	100%	75%	-
jmonkey	FP	8	0	0	12	-	59	32	0	31	-	0	-	0	-	-
	Precision	95%	100%	100%	95%	-	96%	88%	100%	89%	-	100%	-	100%	-	-
	FN	75	0	0	86	-	13	7	0	0	-	0	-	0	-	-
	Recall	65%	100%	100%	72%	-	99%	97%	100%	100%	-	100%	-	100%	-	-
jna	FP	5	0	0	-	-	43	-	-	0	-	0	0	-	-	2
	Precision	84%	100%	100%	-	-	88%	-	-	100%	-	100%	100%	-	-	78%
	FN	22	0	0	-	-	64	-	-	1	-	2	0	-	-	2
	Recall	54%	100%	100%	-	-	83%	-	-	83%	-	71%	100%	-	-	78%
Average	Precision	94%	99%	98%	94%	-	94%	72%	-	81%	-	100%	-	100%	98%	92%
	Recall	81%	98%	98%	86%	-	88%	90%	-	91%	-	88%	-	100%	76%	79%