Multi-language Design Smells: Characteristics, Prevalence, and Impact

- Ph.D. Dissertation -

Mouna Abidi

Supervisor: Prof. Foutse Khomh

Computer Engineering and Software Engineering Department – Polytechnique Montreal

May 5th, 2021

SWAT

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

What is a Multi-language System?

Multi-language Systems

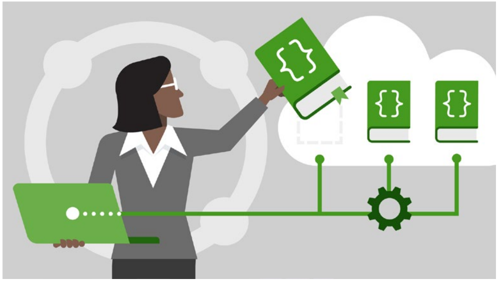
Benefits of Multi-language Systems

Lower development cost

Save development time

Choose programming language

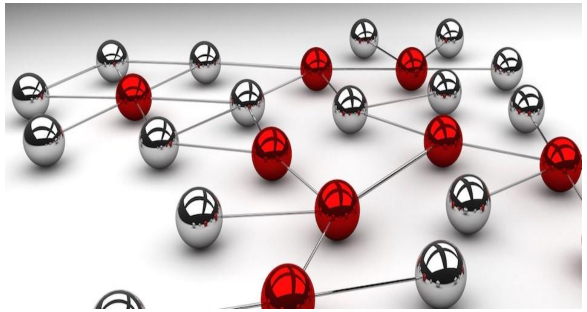
Reuse of code



Reuse of libraries

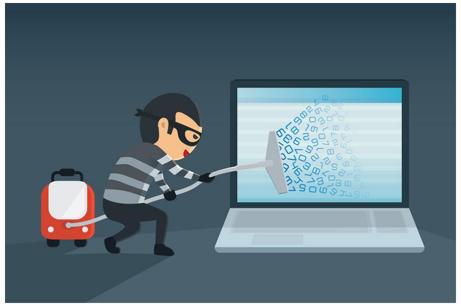
Limitations of Multi-language Systems

Complex interactions



Dependency issues

Higher maintenance cost



Security issues

Hard to understand

Additional bugs

Issues Related to Multi-language Systems

Apparent Facebook Widget Snafu Brings Down Sites

Error

An error occurred. Please try again later.

Ok

THIS IS WHAT PEOPLE ACROSS THE WEB SAW ON THURSDAY WHEN TRYING TO REACH MANY SITES TH
FACEBOOK WIDGETS.

SEVERAL SITES ACROSS the web could not be reached by so
visitors on Thursday afternoon, apparently because of a probler
with Facebook widgets embedded in the sites. Several sites —
including Business Insider, Huffington Post and Salon — were
reportedly affected, redirecting visitors to a Facebook error pag

Facebook did not immediately respond to a request for comme
the problem has apparently been fixed. The problem was first
reported by Marketing Land.

When trying to visit a page that used Facebook Connect or Like widgets, users were redirected to a page saying simply "An error occurred. Please try again later." When they clicked the "Okay" button, they were taken to an error page. If they hit back, they would get to the page they were trying to visit momentarily before being automatically forwarded to the error page again.

Facebook provides code to embed widgets that display information such as which of your friends like a site's Facebook page, or which articles have recently been "liked" by a friend. These widgets execute JavaScript code in the user's web browser that originates at Facebook, not the site that the user is trying to view. The problem only seems to affect users who are not logged into Facebook.

JNI UnsatisfiedLinkError issue

Asked 5 years, 2 months ago Active 5 years, 2 months ago Viewed 172 times

0

I am creating a Java program using JNI to gather data via a C program. I have gone through this JNI tutorial (https://thenewcircle.com/static/bookshelf/java_fundamentals_tutorial/_java_native_interface_jni.html), and everything compiles correctly. However, when I try to run the Java program in Eclipse, I keep getting this error:

Exception in thread "main" java.lang.UnsatisfiedLinkError: no TurtleTrackerImpl in java.library.path at

- [JRUBY-6248] - thread leak
- [JRUBY-6250] - When executing an Ant build.xml file, the Ant executable should not be required to live on the environment's $PATH
- [JRUBY-6251] - NailGun and 1.9 seem not to be usable at the same time (--1.9 and --ng)
- [JRUBY-6259] - ant test - fails in WinXP: (LoadError) no such file to load -- jruby
- [JRUBY-6265] - Setting load path on ScriptingContainer with LocalContextScope.SINGLETON does not work
- [JRUBY-6266] - Unicode encoding problem in CSV.foreach
- [JRUBY-6269] - JRuby --1.9 cannot load YAML output from JRuby --1.8
- [JRUBY-6277] - Dependency to compiler package from org.jruby.Ruby breaks Ruboto
- [JRUBY-6278] - [dev only] Double require bug in the handling of concurrent requires
- [JRUBY-6279] - Invokedynamic support is missing 'float_op_equal'
- [JRUBY-6280] - Fails to open fifo for writing.
- [JRUBY-6281] - [1.9] Applet does not work in the 1.9 mode
- [JRUBY-6282] - Colon is not allowed in a file name on Windows
- [JRUBY-6283] - Master crashes when calling an FFI-attached C library function
- [JRUBY-6284] - Calls to Kernel#exit result in an exception printed on stderr
- [JRUBY-6285] - JRuby 1.7 master on Java7u2 is *slower* running a benchmark than master on Java6
- [JRUBY-6291] - Closing One Stream From IO.popen4 Results in Stream Closed Error When Reading Other Streams
- [JRUBY-6292] - Massive perf degradation in pack after ByteList update
- [JRUBY-6293] - jruby-dist-master does not build C extensions
- [JRUBY-6295] - Dir.chdir, $HOME and $LOGDIR behavior
- [JRUBY-6300] - TestMethodmissing testcase fails with Java 7
- [JRUBY-6301] - scripting_lang.jruby:undefined method in test_loop_1_9.rb
- [JRUBY-6305] - C Extension fails to build

ANN] JRuby 1.7.0.preview1 released

ruby-talk

- Java 5 support dropped (Java 6+ required)
- Update to Rubygems 1.8.24
- Update to Rake 0.9.2.2
- 259 issues resolved

*Note on invokedynamic performance:

Invokedynamic is still a new feature for the JVM, so we recommend
ild of Java 7 as possible. Builds of
prior to "update 2" will show poor performance.

t can be disabled with
mic=false (passed to JRuby) for investigating perf

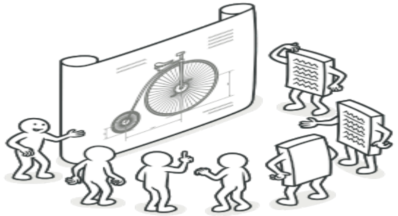
ays don't inherit from java.lang.Object in
erarchy
ting is slower in JRuby than MRI
should load relative path reference to AOT classes
reopen a class from an included module
work replacement char with russian charset
_class should be deprecated in favor of
port)
:gethostbyname does reverse DNS lookup for IP
no DNS reverse lookup reply is received
onsistency handles file:/// URLs pointing to

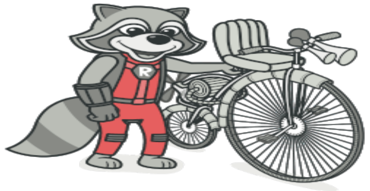
ec: SimpleDelegator send spec fails due to bug in

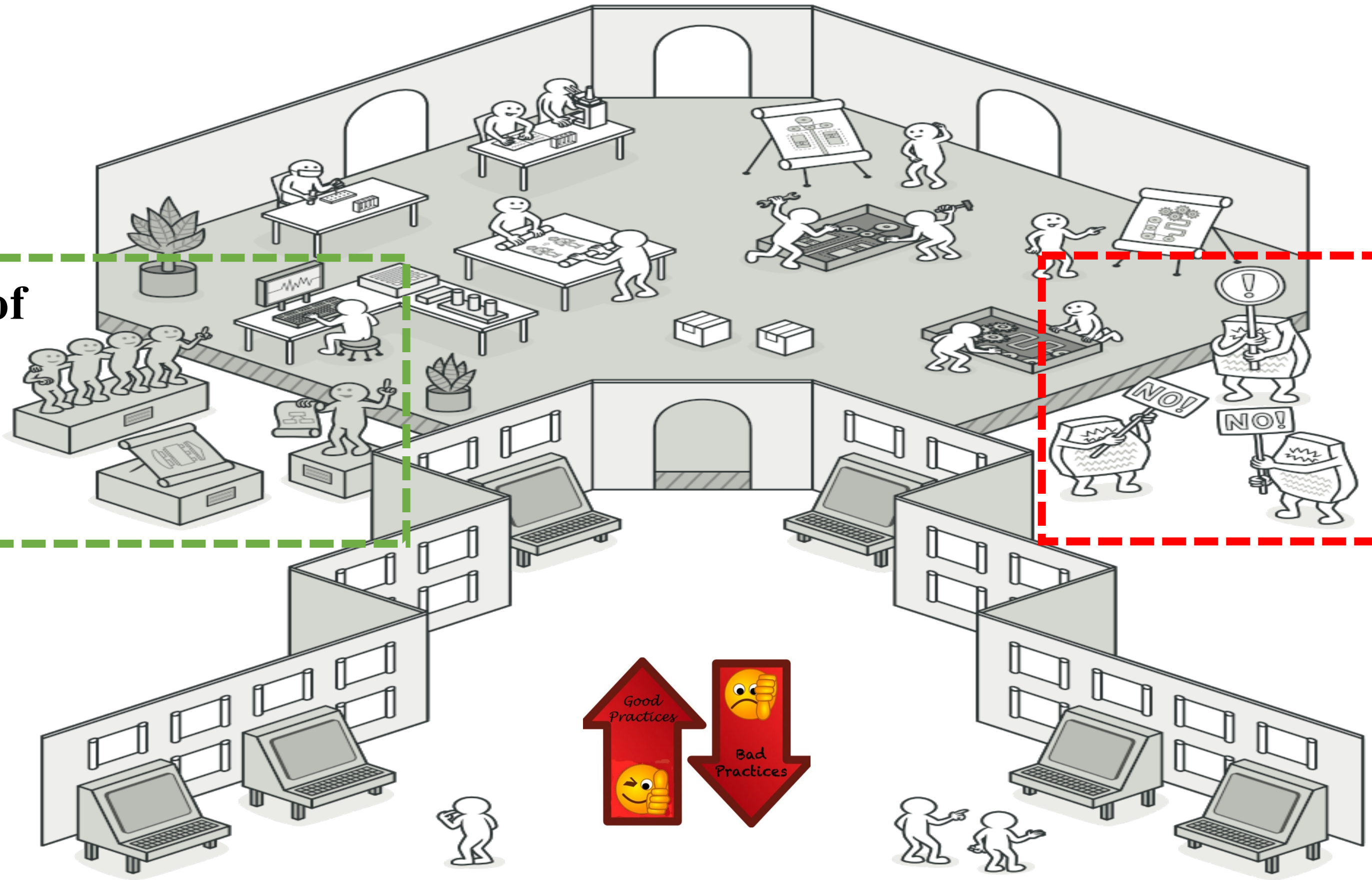
b:24: superclass must be a Class (Module given) (TypeError)
name(__FILE__) doesn't return correct value when
classpath
reopen JRUBY-3894
erver#accept can't be interrupted by kill/raise
load with wrap=true does not protect the global
program
tracer doesn't trace
#select puts connectable sockets in the read set

w digest methods are missing (base64digest and

Design Smells







Identification of good practices and design patterns

Identification of bad practices and design smells

https://refactoring.guru/design-patterns

Literature

2012 28th IEEE International Conference on Software Maintenance (ICSM)

Piecemeal Migration of a Document Archive System with an Architectural Pattern Language

Build System Issues in Multilanguage Software

Ar
Depa

Finding Bugs in Java Native Interface Programs

Goh Kondoh Tamiya Onodera
Tokyo Research Laboratory
IBM Researcl
1623-14, Shimotsuruma,
Kanagawa-ken, J
+81-46-215-4584, +81-4
{gkondo,tonodera}@j

ABSTRACT

In this paper, we describe static analysis techniques for finding bugs in programs using the Java Native Interface (JNI). The JNI is both tedious and error-prone because there are many JNI-specific mistakes that are not caught by a native compiler. This paper is focused on four kinds of common mistakes. First, explicit statements to handle a possible exception need to be inserted after a statement calling a Java method. However, such statements tend to be forgotten. We present a typestate analysis to detect this exception-handling mistake. Second, while the native code can allocate resources in a Java VM, those resources must be manually released, unlike Java. Mistakes in resource management cause leaks and other errors. To detect Java resource errors, we used the typestate analysis also used for detecting general memory errors. Third, if a reference to a Java resource lives across multiple native method invocations, it should be converted into a global reference. However, programmers sometimes forget this rule and, for example, store a local reference in a global variable for later uses. We provide a syntax checker that detects this bad coding practice. Fourth, no JNI function should be called in a critical region. If called there, the current thread might block and cause a deadlock. Misinterpreting the end of the critical region, programmers occasionally break this rule. We present a simple typestate analysis to detect an improper JNI function call in a critical region.

We have implemented our analysis techniques in a bug-finding tool called BEAM, and executed it on opensource software including JNI code. In the experiment, our analysis techniques found 86 JNI-specific bugs without any overhead and increased the total number of bug reports by 76%.

Ge
Algo
Lang

Key
Java

1.
A fo
lang
prog
versi
JNI
prog
Pitfa
To e
term
Also
resul
with
com
whil
[3].
pars
C++
prob
JNI
Tabl
prac
beyo

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2002; **14**:1–30 (DOI: 10.1002/smr.243)

Research

Piecemeal legacy migrating with an architectural pattern language: a case study

M. Goedicke and U. Zdun*,†

Specification of Software Systems, University of Essen, Germany

SUMMARY

Numerous large applications that have evolved over many years are well-functioning and reliable, but have severe problems regarding flexibility and reuse. Due to the many fixes that were applied in a system's lifetime, it is often hard to customize, change or exchange system parts. Therefore, it is problematic to migrate such systems to a more flexible architecture or to new technologies. The document archive/retrieval system, discussed in this article, is an example of a large C system that had such problems. As a solution, we will sketch an architectural pattern language that involves patterns well-suited for a piecemeal migration process. The patterns aim at building and composing highly flexible black-box component architectures with an object-oriented glueing layer. We present a re-engineering case study for the document archive/retrieval system based on these patterns. The patterns are used to wrap the existing C implementations and integrate them with an object system. Moreover, the patterns introduce flexibility hooks into the hot spots of the architecture and let components define their required environment. This enables an easier future evolution of the system. The case study demonstrates a pattern language as an approach for piecemeal legacy migration apart from implementation details. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: software pattern; pattern language; re-engineering; component architecture

Finding Bugs in Exceptional Situations of JNI Programs

Siliang Li
tment of Computer Science and Engineering
Lehigh University
sil206@cse.lehigh.edu

Gang Tan
Department of Computer Science and Engineering
Lehigh University
gtan@cse.lehigh.edu

tive methods may defeat Java's guaran-
urity. One common kind of flaws in na-
from the discrepancy on how exceptions
and in native methods. Unlike excep-
tions raised in the native code through
erface (JNI) are not controlled by the
ne (JVM). Only after the native code
ill the JVM's mechanism for exceptions
repancy makes handling of JNI excep-
process and can cause serious security
itten using the JNI.
rel static analysis framework to exam-
eport errors in JNI programs. We have
l consisting of exception analysis, static
warning recovery. Experimental results
ool allows finding of mishandling of ex-
accuracy (15.4% false-positive rate on
ode). Our framework can be easily ap-
ftware written in other foreign function
the Python/C interface and the OCam-

Subject Descriptors

oftware Engineering—*Software/Program*
[**Software**]: Software Engineering—*In-*

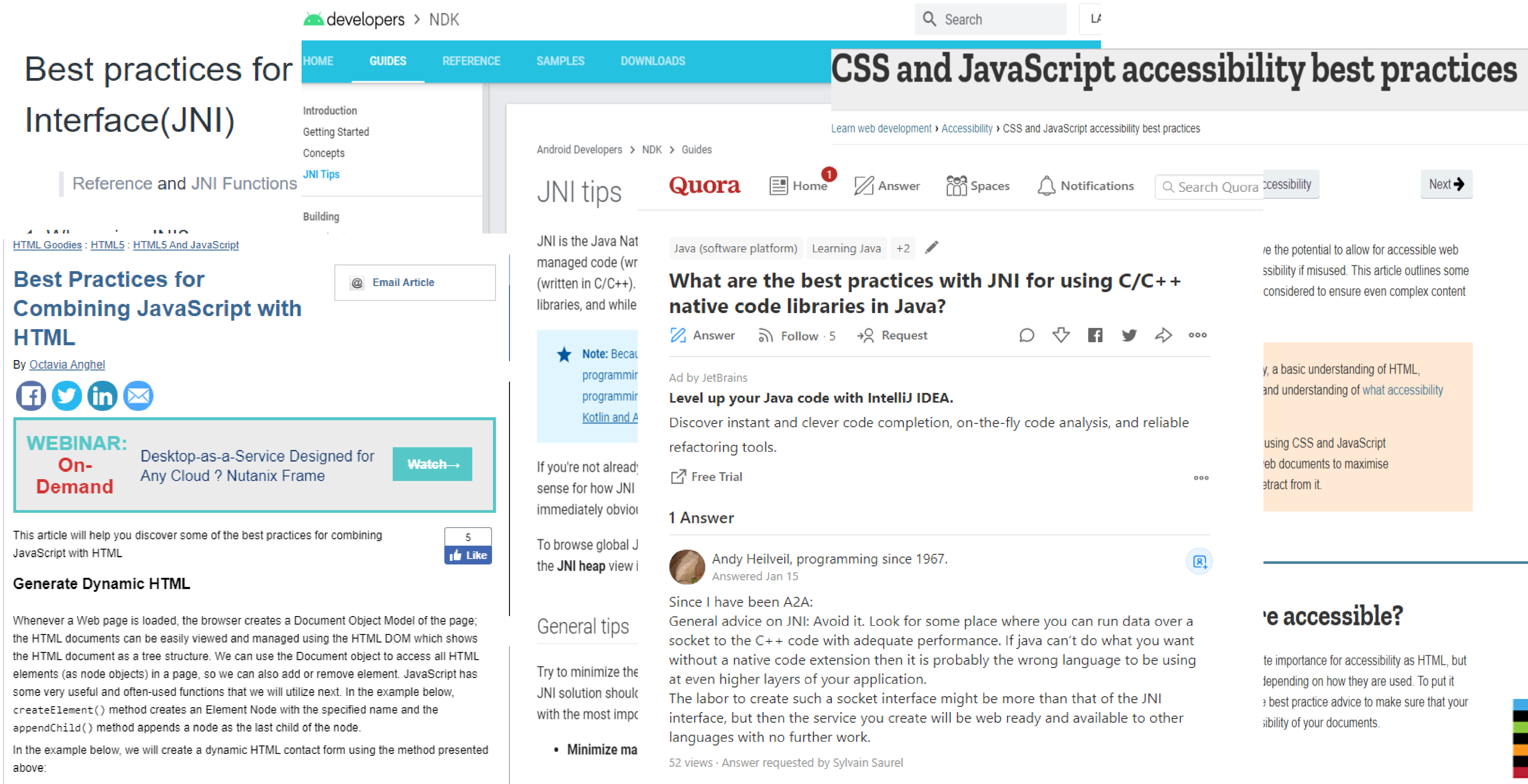
much less vulnerable. As another example, Perl's taint mode prevents attacks based on malicious user input. In both cases, managed environments provide a natural and extensible way of enforcing relevant security policies.

To interoperate with software components in other languages, most managed programming languages also support foreign function interfaces (FFIs). The Java Native Interface (JNI) allows Java components to interoperate with native components developed in C, C++, or assembly languages. Similarly, .NET provides the P/Invoke interface for invoking library functions.

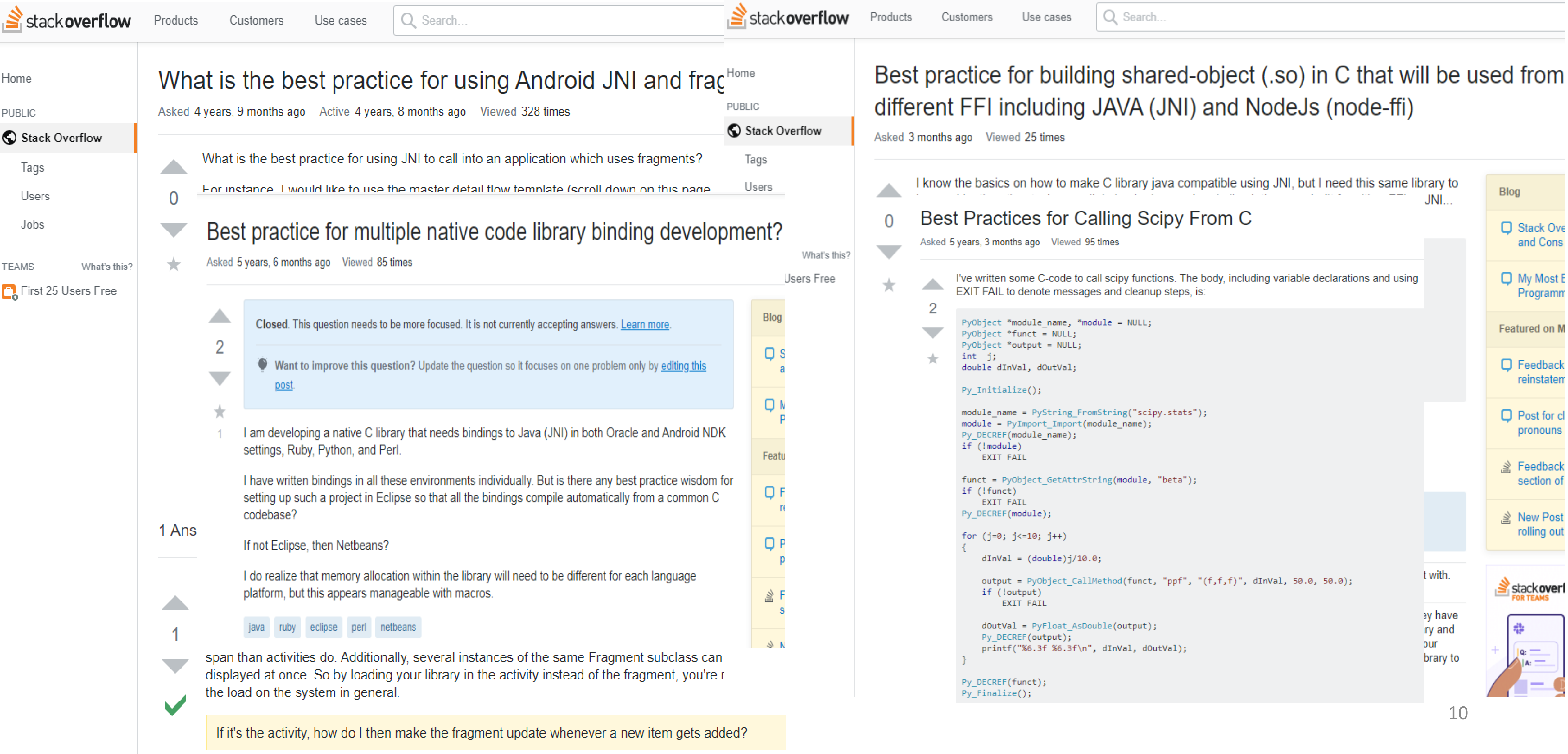
Native components are usually the security dark corner of software applications. They are outside of managed environments and relevant security policies cannot be enforced on them. In Sun's JDK 1.6, there are over 800,000 lines of C/C++ code.[1] Any vulnerability in this trusted native code can compromise the security of the JVM. Several vulnerabilities have been discovered [24, 30, 29]. A recent empirical security study [28] on Sun's JDK 1.6 found over 126 software errors in a mere 38,000 lines of C code. 59 of them are security critical.

One of the most revealing aspects of the security study is that many of the discovered errors are due to a discrepancy on how exceptions are handled between Java and the JNI. Managed environments such as the JVM provide runtime support for exception handling, which native components cannot rely on. We next explain why this discrepancy may lead to security vulnerabilities and why it is common in foreign function interfaces.

Developers’ Blogs



Developers' Blogs



Design patterns books

Google

design patterns

- design patterns
- design patterns **java**
- design patterns **book**
- design patterns **c#**
- design patterns **python**
- design patterns **interview questions**
- design patterns **programming**
- design patterns **mvc**
- design patterns **list**
- design patterns **wiki**

Google

design patterns for multi-language

- design **pattern multi language**
- **database** design **pattern multi language**

Google Search | I'm Feeling Lucky

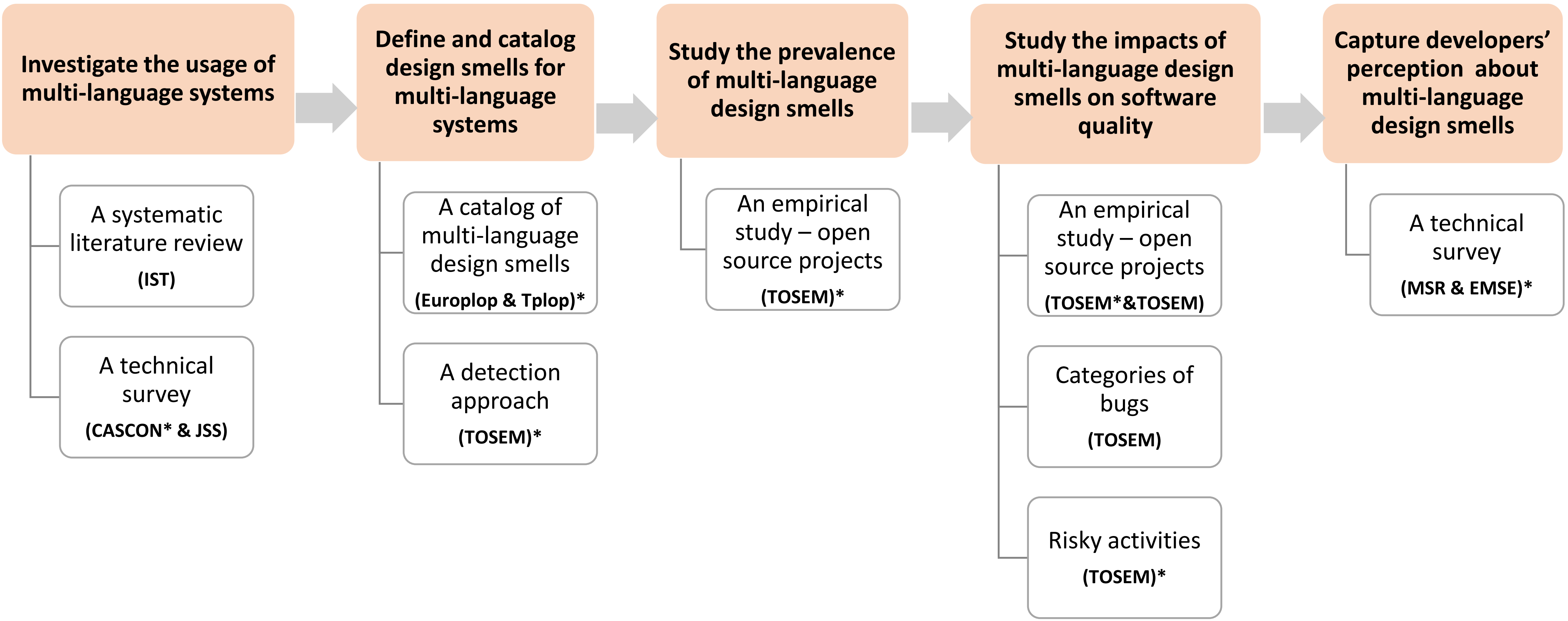
Thesis Statement

- **Design smells exist** in multi-language systems **(H1)**
- Multi-language design smells are **prevalent** in open source projects **(H2)**
- Multi-language design smells present **negative impacts** on the software quality **(H3)**

Objectives

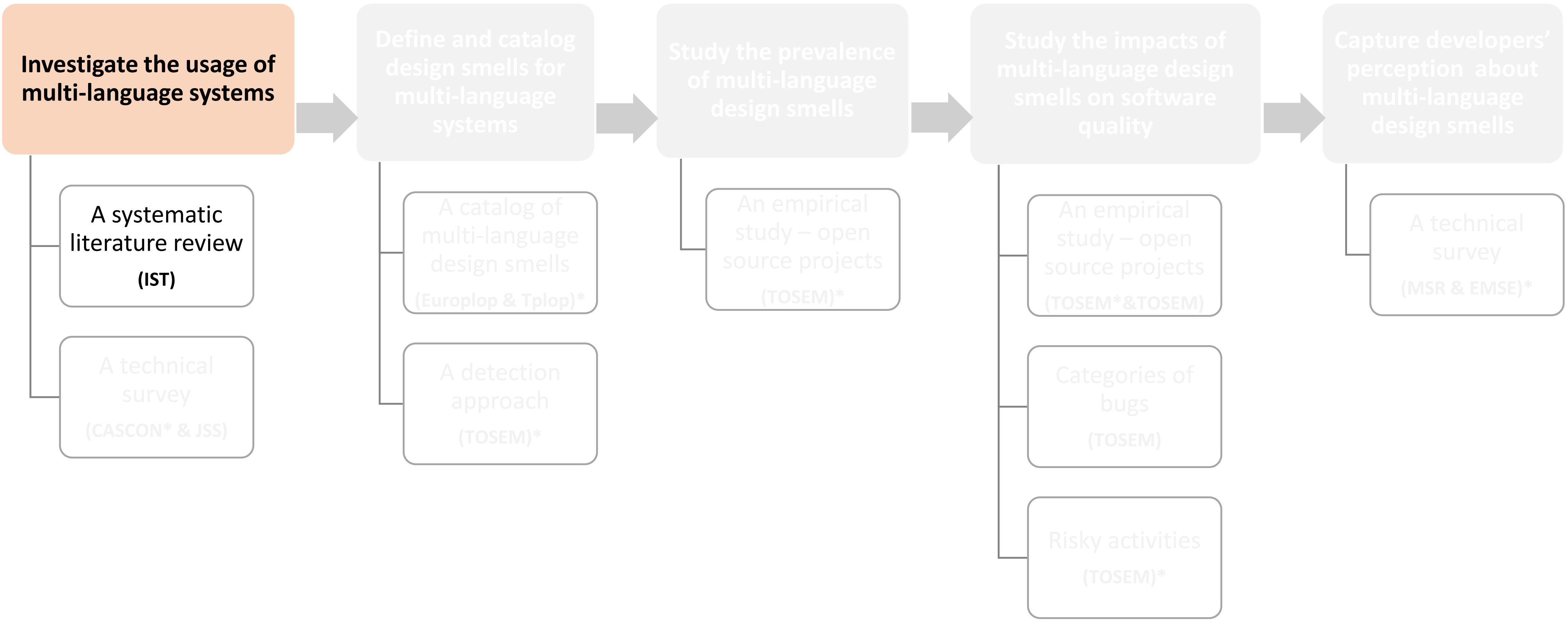
- Define and **catalog** design **smells** for **multi-language systems**
- Study the **prevalence** of multi-language design **smells**
- Study the **impacts** of multi-language design **smells** on **software quality**

Thesis Overview



* Accepted papers

Thesis Overview



* Accepted papers

Pilot 1 - Systematic Literature Review

Query 3:
((((((((((((((((('multiple languages' OR 'multiple language' OR multi-language* OR 'multi language' OR 'multi languages' OR mixed-language* OR 'mixed language' OR 'mixed languages' OR 'heterogeneous language' OR 'heterogeneous languages' OR polylingual OR polyglot)wn KY AND (software* OR Program* OR Analys* OR System*)wn KY))))))))))))))) AND ((2020 OR 2019 OR 2018 OR 2017 OR 2016 OR 2015 OR 2014 OR 2013 OR 2012 OR 2011 OR 2010) wn YR)) AND (english wn LA))

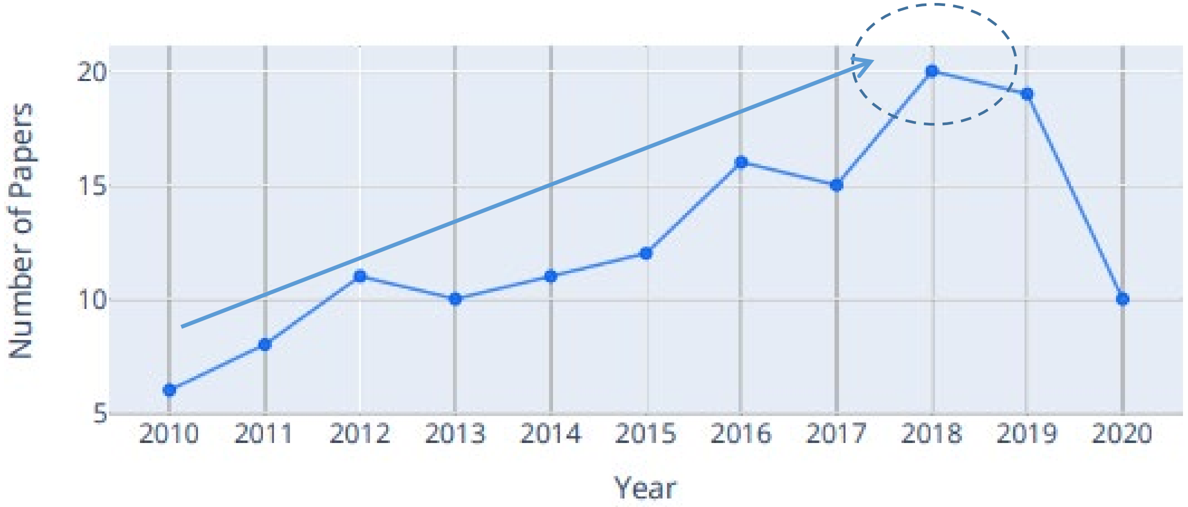
3694 papers

Inclusion and Exclusion Criteria

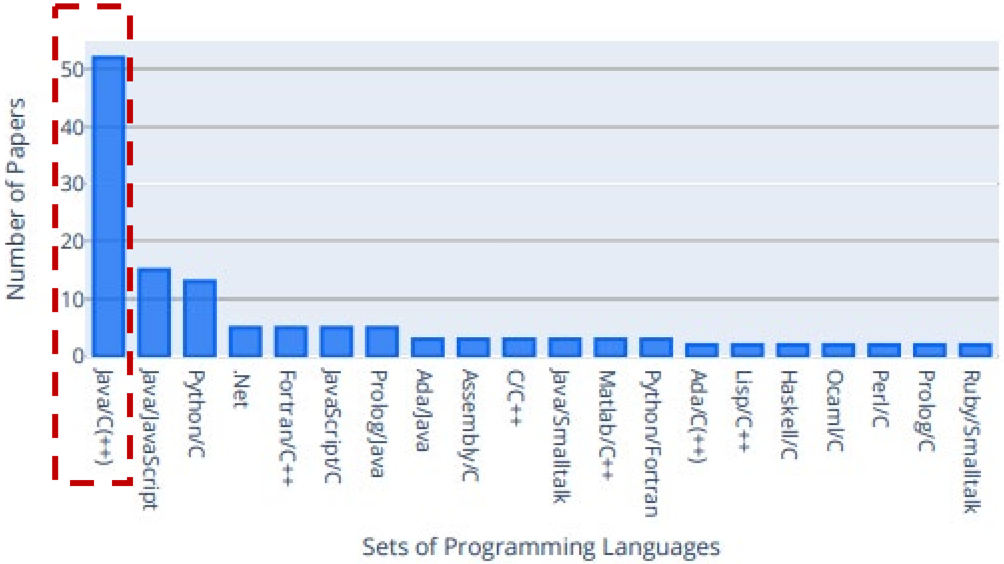
138 papers

Data Extraction

Study Results

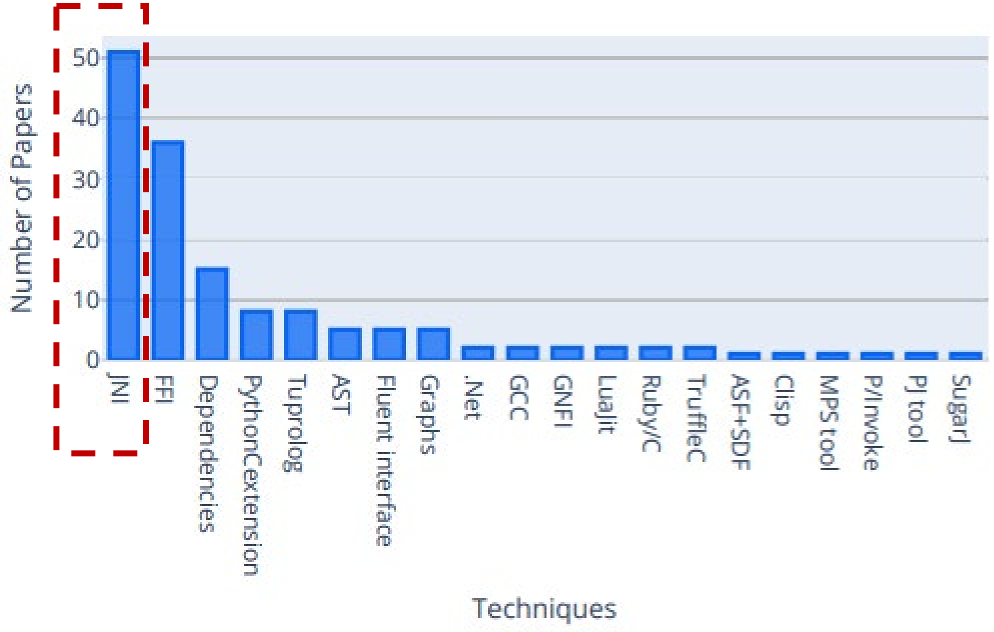


Multi-language Papers Over Time

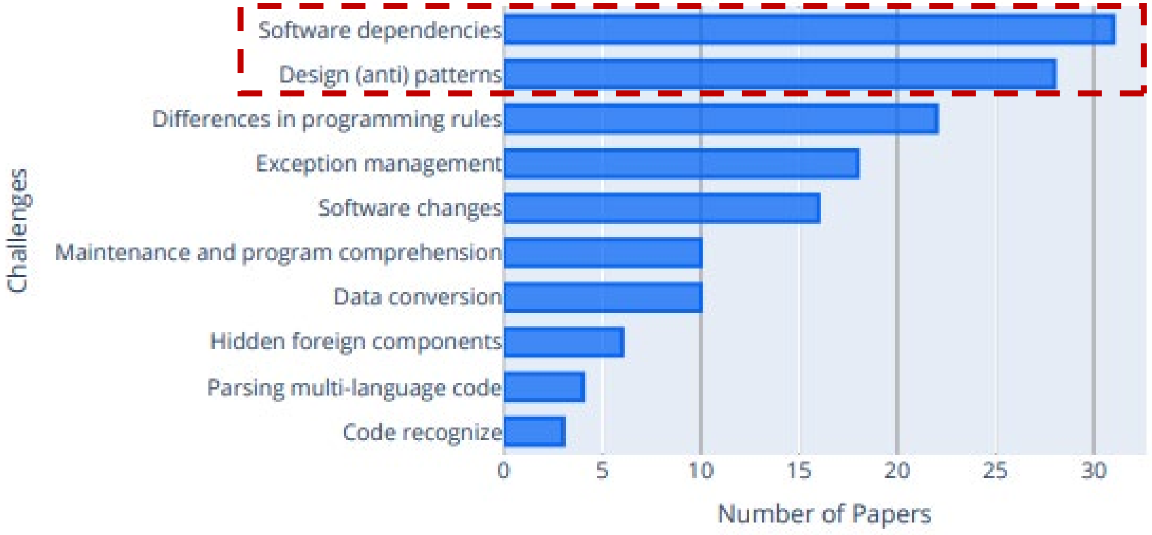


The Top 20 Combinations of Programming Languages Discussed in Literature

Study Results

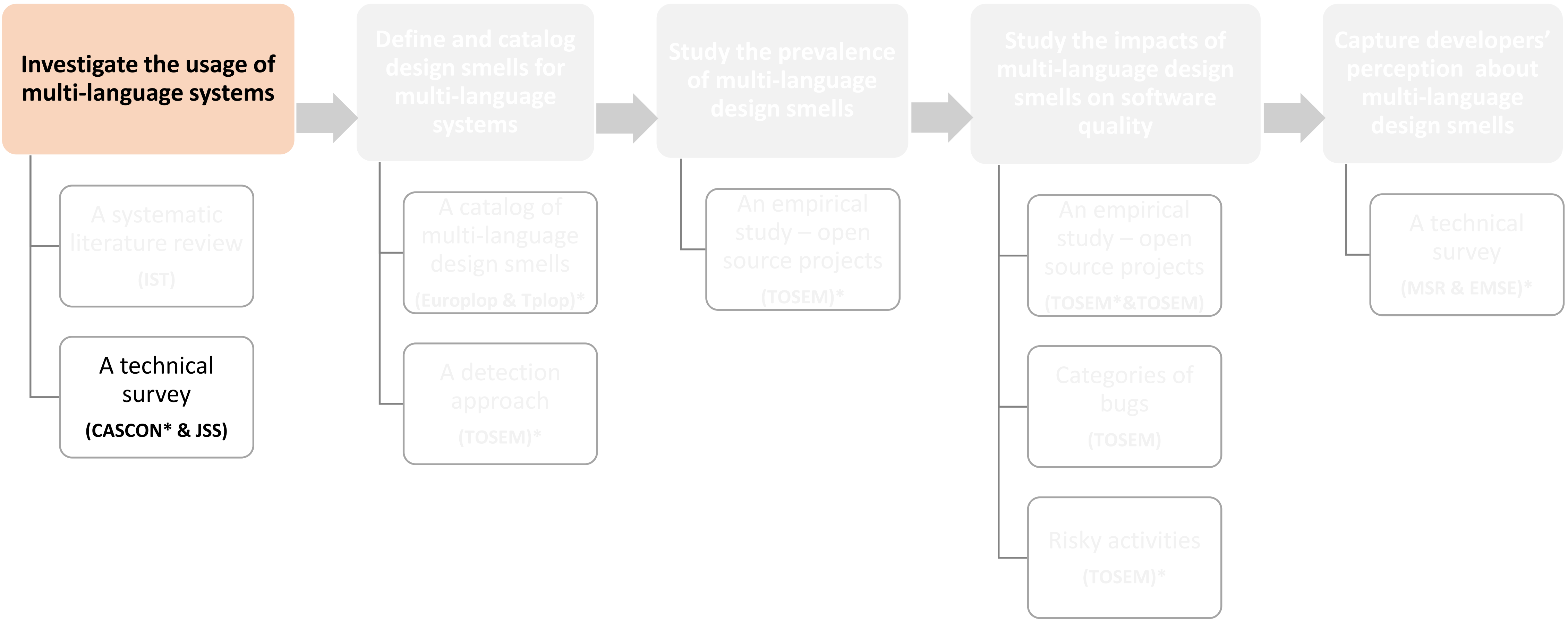


Techniques Used for the Integration of Programming Languages



Major Challenges of Multi-language Systems

Thesis Overview



* Accepted papers

Pilot 2 – Technical Survey

Study Design

Survey

Participants collection

Survey Administration

Data Analysis

133 participants (47.5%)

Developers' Perspectives on Multi-language Systems

- **Increasing popularity**

- **Perceived benefits**:
  - ➢Ease implementation of the initial code
  - ➢Reuse of code
  - ➢Benefits from each programming language
  - ➢Increase developers' motivation

"Good practices and tools for multiple language may help developers keep their code clean and maintainable" (Participant)

- **Perceived Challenges**:
  - ➢ Complex maintenance
  - ➢ Diverse competences requirements
  - ➢ Complex dependencies
  - ➢ Lack of dedicated support

- **Current Solution:**
  - ➢ Mono-language patterns and solutions for multi-language systems

Implications from the Pilot Studies

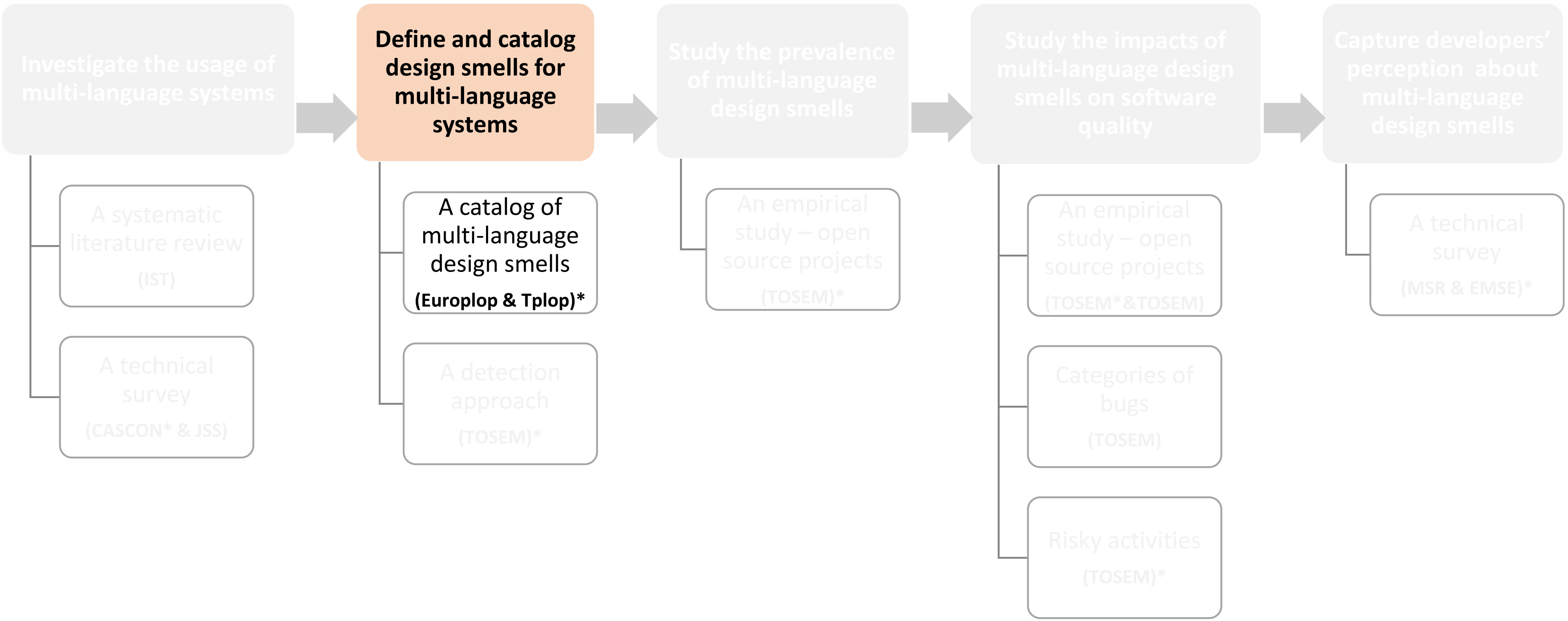
Information scattered

Concrete relevance

Evaluation of impact

Developers’ perception

Thesis Overview



* Accepted papers

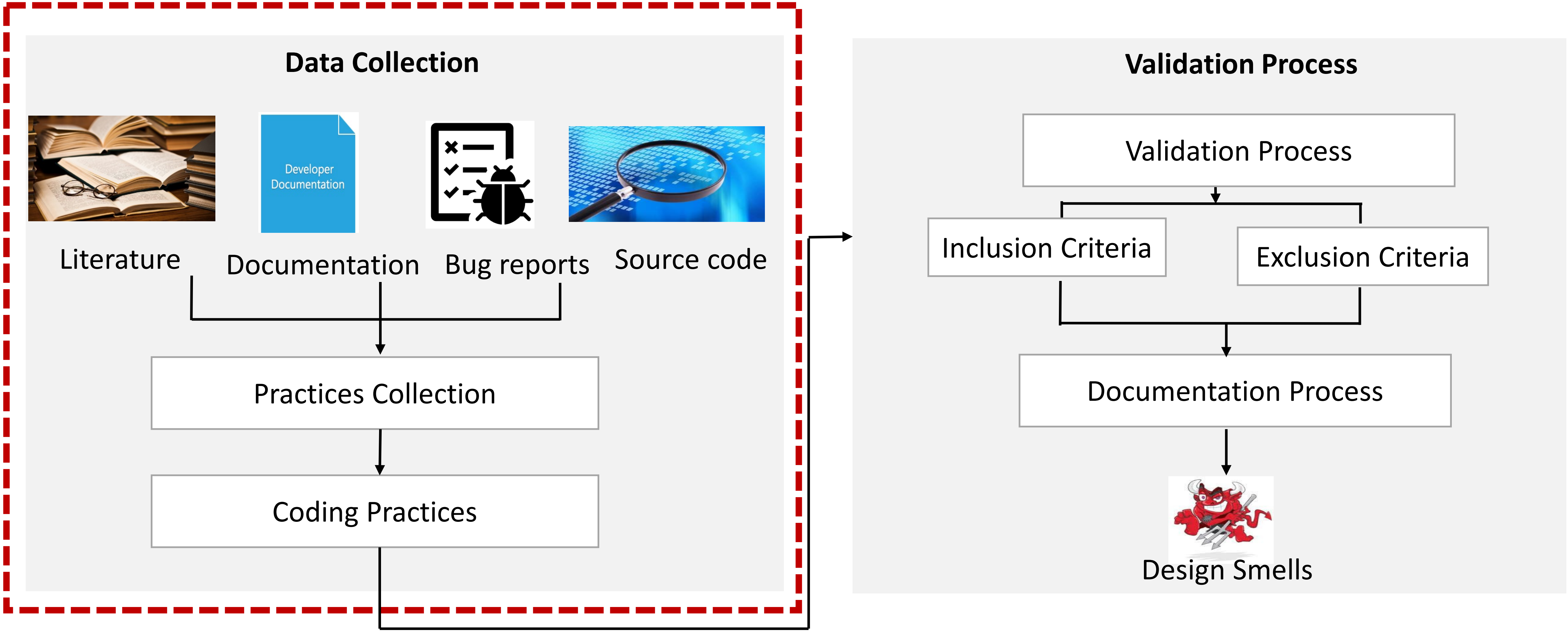
Multi-language Design Smells

- **Multi-language design smells** are defined as **poor design** and **coding decisions** when **bridging** between **different programming languages**

- Design smells include anti-patterns and code smells

- They represent **violations** of **best practices** related to the **combination of programming languages** that often indicate the presence of bigger problems

Study Design



Examples of Collection of Practices

Bad Practices

Good Practices

Error handling

Using native methods in Java programs breaks the Java security model in s
ways. Because Java programs run in a controlled runtime system (the JVM)
the Java platform decided to help the programmer by checking common ru
array indices, out-of-bounds errors, and null pointer errors. C and C++, on t
no such runtime error checking, so native method programmers must hand
conditions that would otherwise be caught in the JVM at runtime.

For example, it is common and correct practice in Java progran
throwing an exception. C has no exceptions, so instead you mu
functions of JNI.

JNI's exception handling functic

There are two ways to throw an exception in the native code: y
or the ThrowNew() function. Before calling Throw(), you first n
Throwable. By calling ThrowNew() you can skip this step becau
object for you. In the example code snippet below, we throw ar
functions:

```
/* Create the Throwable object. */
jclass cls = (*env)->FindClass(env, "java/io/IOExceptio
jmethodID mid = (*env)->GetMethodID(env, cls, "<init>",
jthrowable e = (*env)->NewObject(env, cls, mid);

/* Now throw the exception */
(*env)->Throw(env, e);
...

/* Here we do it all in one step and provide a message*
(*env)->ThrowNew(env,
                 (*env)->FindClass("java/io/IOException
                 "An IOException occurred!");
```

2. Performance pitfalls

- **Not caching method IDs, field IDs, and Classes**

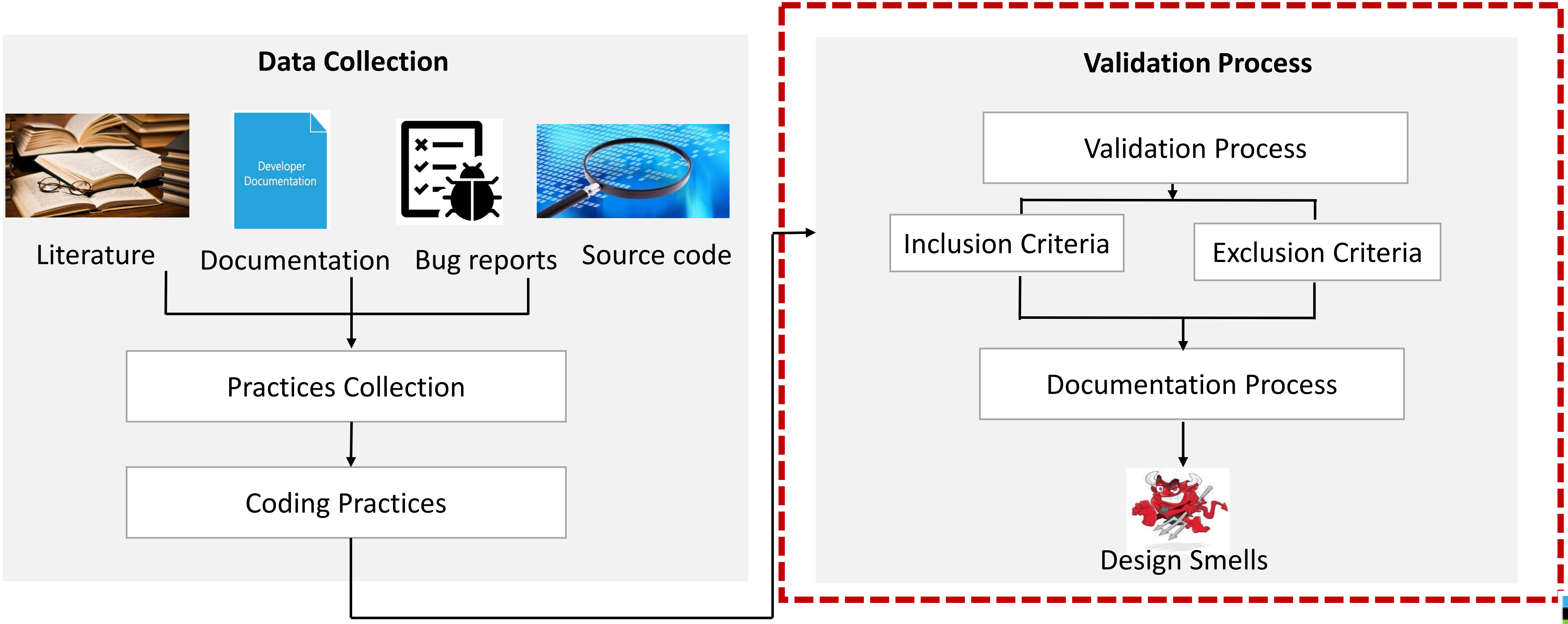
To access Java objects' fields and invoke their methods, native code must make calls to `FindClass()`, `GetFieldID()`, `GetMethodId()`, and `GetStaticMethodID()`. The IDs returned for a given class don't change for the lifetime of the JVM process. But the call to get the field require significant work in the JVM. Because the IDs are the same for a given class, you should look them up once and then reuse them.

General tips

Try to minimize the footprint of your JNI layer. There are several dimensions to consider here. Your JNI solution should try to follow these guidelines (listed below by order of importance, beginning with the most important):

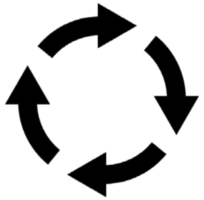
- **Minimize marshalling of resources across the JNI layer.** Marshalling across the JNI layer has non-trivial costs. Try to design an interface that minimizes the amount of data you need to marshall and the frequency with which you must marshall data.
- **Avoid asynchronous communication between code written in a managed programming language and code written in C++ when possible**. This will keep your JNI interface easier to maintain. You can typically simplify asynchronous UI updates by keeping the async update in the same language as the UI. For example, instead of invoking a C++ function from the UI thread in the Java code via JNI, it's better to do a callback between two threads in the Java programming language, with one of them making a blocking C++ call and then notifying the UI thread when the blocking call is complete.
- **Minimize the number of threads that need to touch or be touched by JNI.** If you do need to utilize thread pools in both the Java and C++ languages, try to keep JNI communication between the pool owners rather than between individual worker threads.
- **Keep your interface code in a low number of easily identified C++ and Java source locations to facilitate future refactors.** Consider using a JNI auto-generation library as appropriate.

Study Design



A Catalog of Multi-language Design Smells

- A catalog of 15 types of Multi-language Design Smells



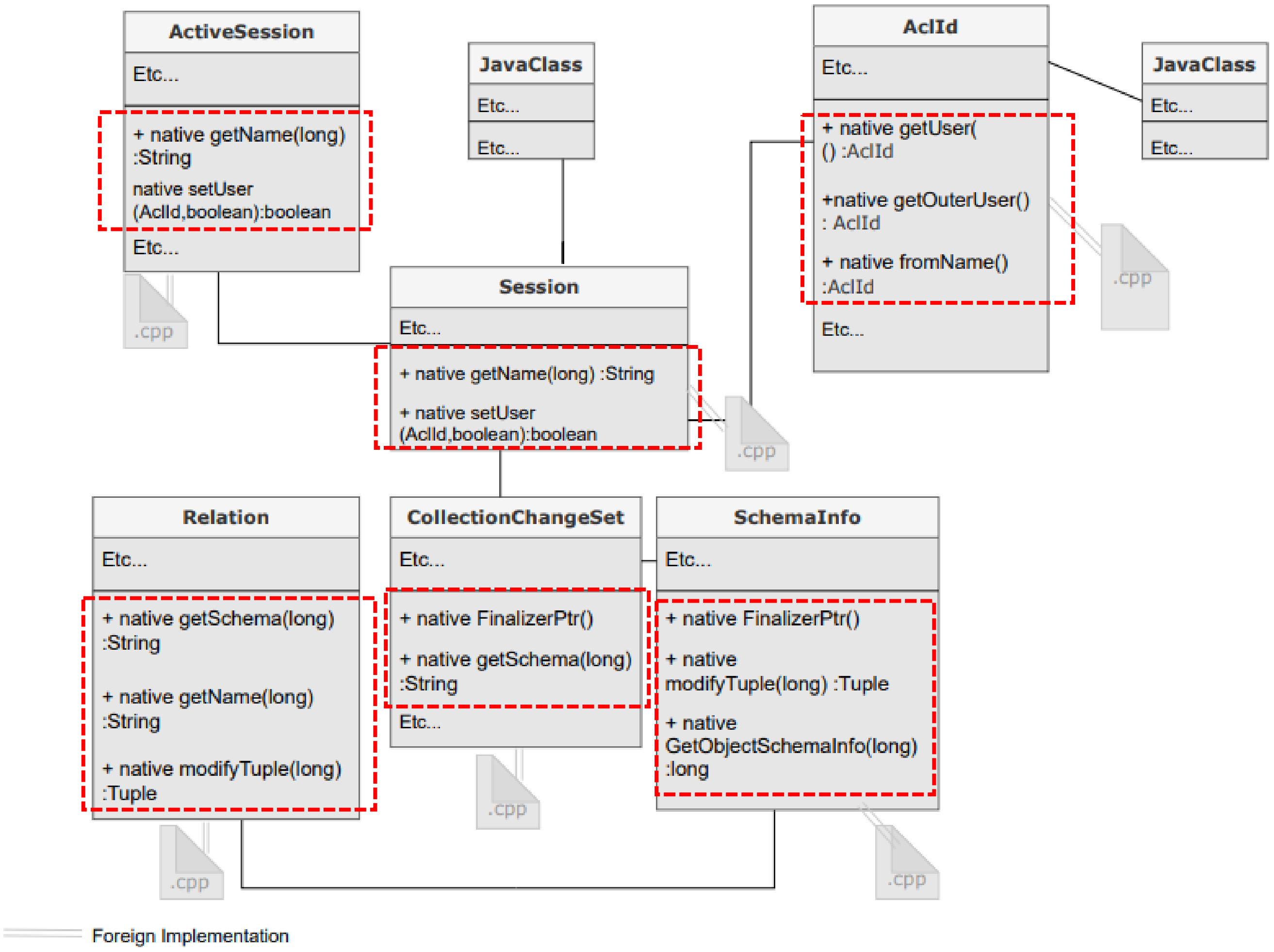
Rounds of shepherding Process

Writers' Workshop

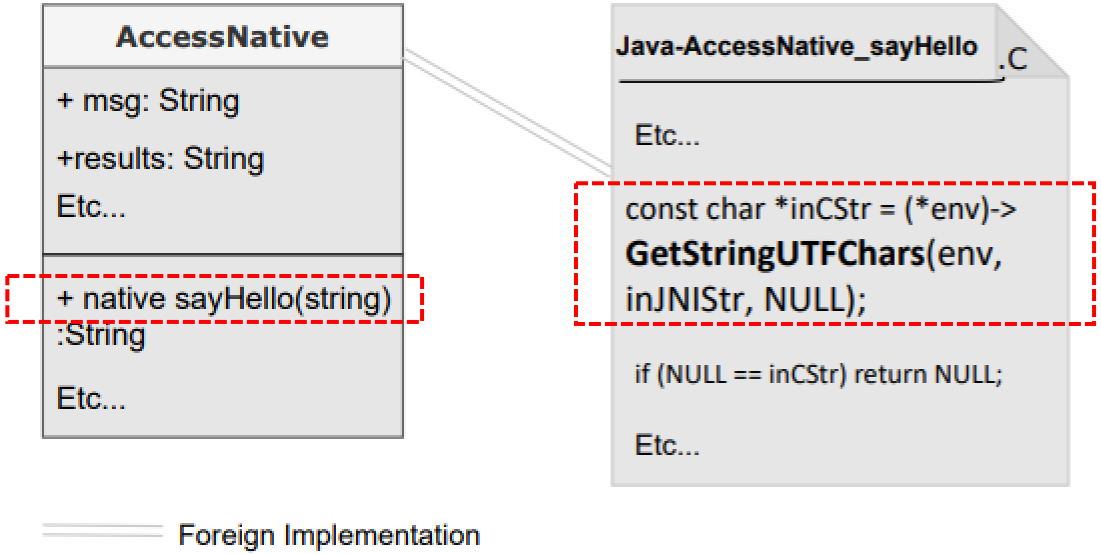
Refine Design Smells

N.	Multi-language Design Smells
1	Not Handling Exceptions
2	Not Securing Libraries
3	Local Reference Abuse
4	Memory Management Mismatch
5	Excessive Objects
6	Too Much Clustering
7	Unused Method Implementation
8	Unused Parameters
9	Assuming Safe Return Values
10	Not Using Relative Path
11	Hard Coding Libraries
12	Not Caching Objects
13	Too Much Scattering
14	Excessive Inter-language Communication
15	Unused Method Declaration

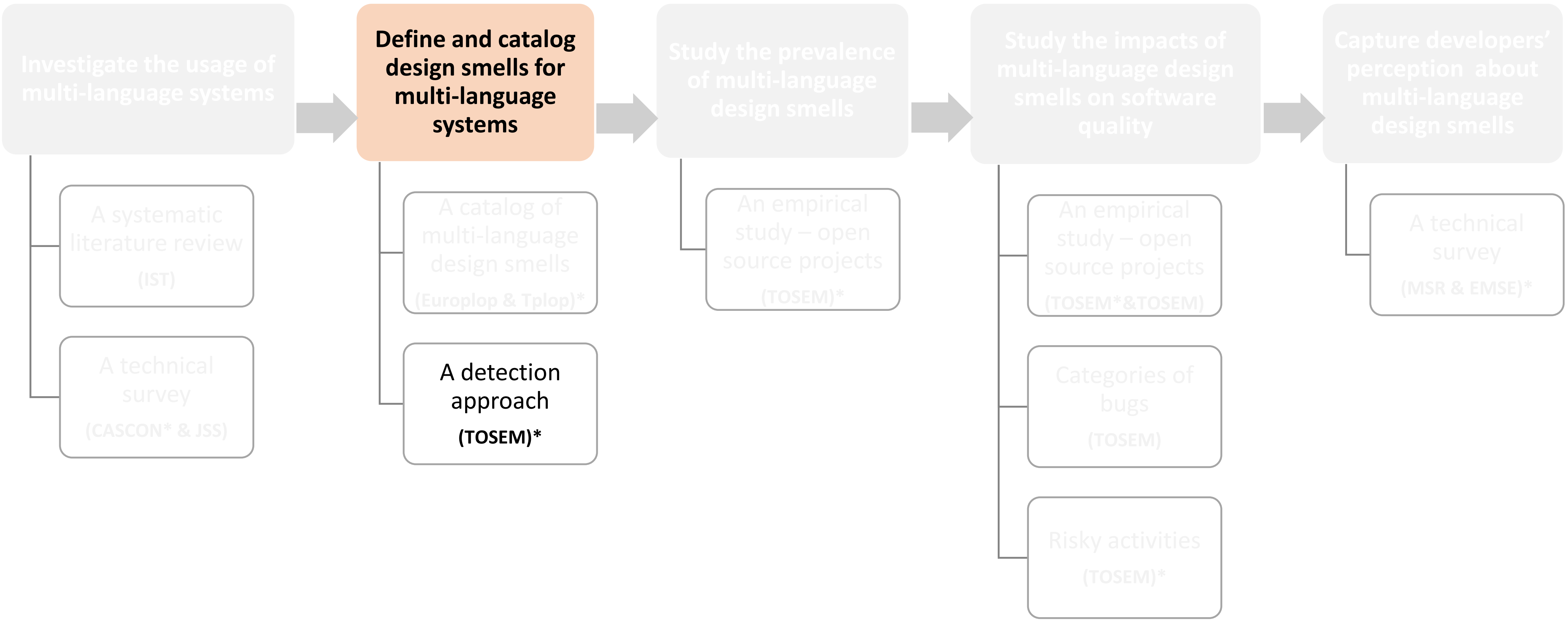
Too Much Scattering



Memory Management Mismatch

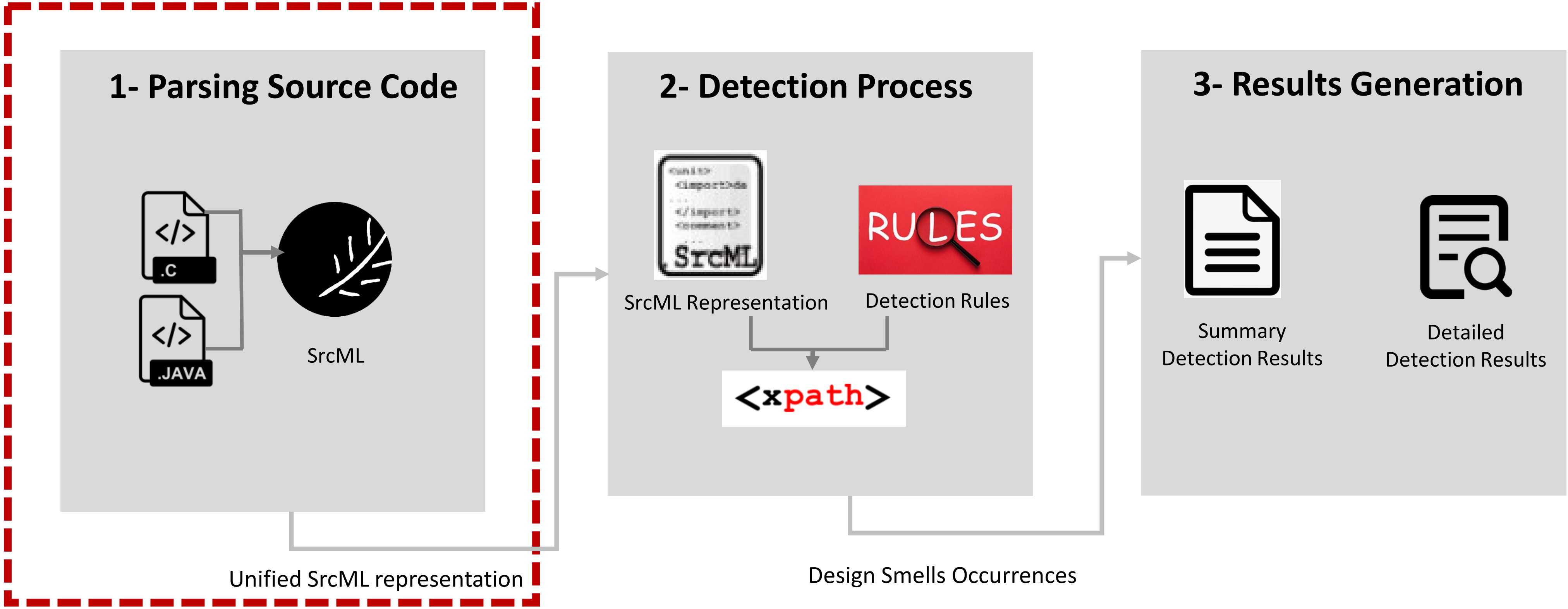


Thesis Overview

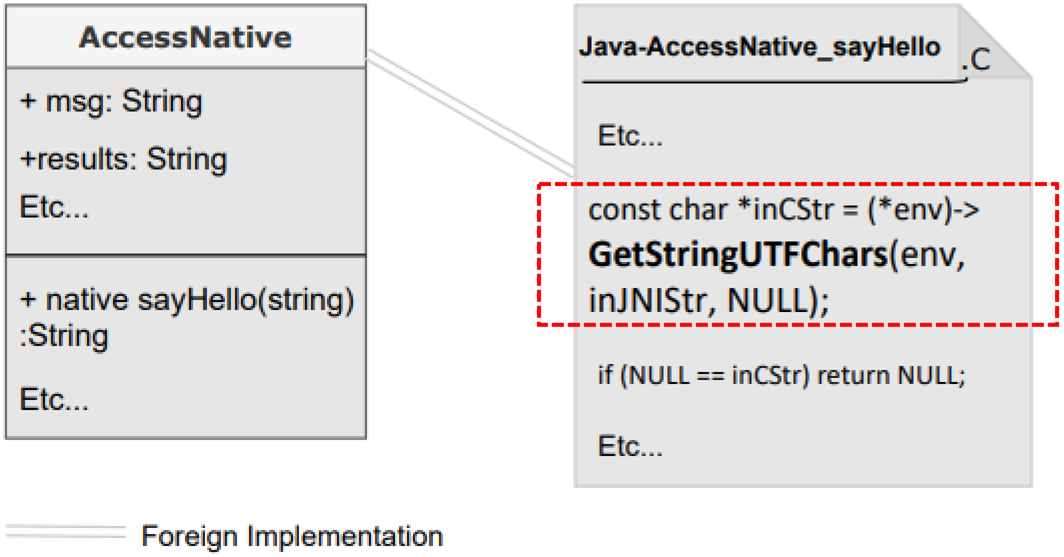


* Accepted papers

MLSInspect: A Detection Approach For Multi-language Design Smells



Memory Management Mismatch

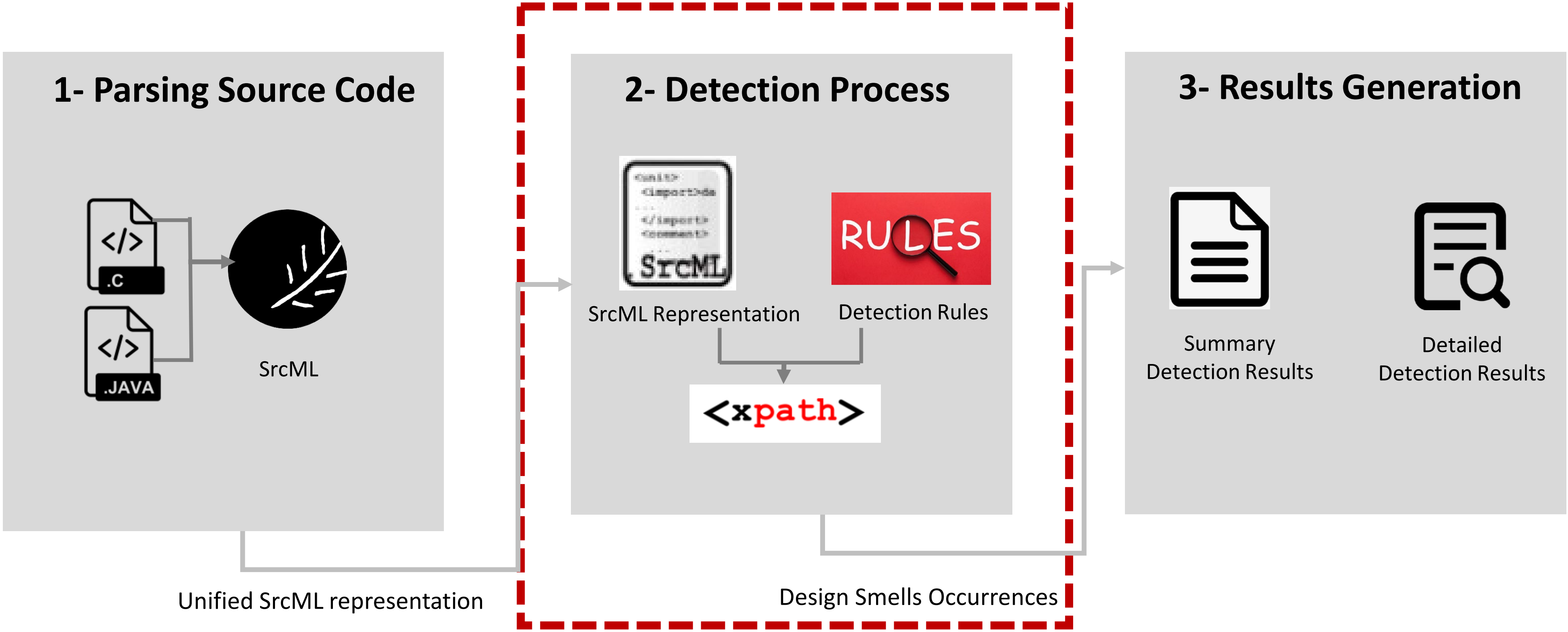


Parsing Source Code

```c
JNIEXPORT jstring JNICALL Java_AccessNative_sayHello(JNIEnv *env, jobject thisObj, jstring inJNIStr) {

   const char *inCStr = (*env)->GetStringUTFChars(env, inJNIStr, NULL);
   if (NULL == inCStr) return NULL;

   printf("In C, the received string is: %s\n", inCStr);

   char outCStr[128];
   printf("Enter a String: ");
   scanf("%s", outCStr);

   return (*env)->NewStringUTF(env, outCStr);
}
```

```xml
<function><type><name>JNIEXPORT</name> <name>jstring</name> <name>JNICALL</name></type> <name>Java_AccessNative_sayHello</name><parameter_list>
(<parameter><decl><type><name>JNIEnv</name> <modifier>*</modifier></type><name>env</name></decl></parameter>, <parameter><decl><type>
<name>jobject</name></type> <name>thisObj</name></decl></parameter>, <parameter><decl><type><name>jstring</name></type> <name>inJNIStr</name>
</decl></parameter>)</parameter_list> <block>{ <decl_stmt><decl><type><specifier>const</specifier> <name>char</name> <modifier>*</modifier></type>
<name>inCStr</name> <init>= <expr><call><name><operator>(</operator><operator>*</operator><name>env</name><operator>)</operator><operator>-&gt;</operator>
<name>GetStringUTFChars</name></name><argument_list>(<argument><expr><name>env</name></expr></argument>, <argument><expr><name>inJNIStr</name></expr>
</argument>, <argument><expr><name>NULL</name></expr></argument>)</argument_list></call></expr></init></decl>;</decl_stmt>
   <if>if <condition>(<expr><name>NULL</name> <operator>==</operator> <name>inCStr</name></expr>)</condition><then> <block type="pseudo"><return>return <expr>
   <name>NULL</name></expr>;</return></block></then></if> <expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">
   "In C, the received string is: %s\n"</literal></expr></argument>, <argument><expr><name>inCStr</name></expr></argument>)</argument_list></call></expr>;
   </expr_stmt> <decl_stmt><decl><type><name>char</name></type> <name><name>outCStr</name><index>[<expr><literal type="number">128</literal></expr>]</index>
   </name></decl>;</decl_stmt><expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">"Enter a String: "</literal></expr>
   </argument>)</argument_list></call></expr>;</expr_stmt><expr_stmt><expr><call><name>scanf</name><argument_list>(<argument><expr><literal type="string">"%s"
   </literal></expr></argument>, <argument><expr><name>outCStr</name></expr></argument>)</argument_list></call></expr>;</expr_stmt><return>return <expr><call>
   <name><operator>(</operator><operator>*</operator><name>env</name><operator>)</operator><operator>-&gt;</operator><name>NewStringUTF</name></name><argument_list
   (<argument><expr><name>env</name></expr></argument>, <argument><expr><name>outCStr</name></expr></argument>)</argument_list></call></expr>;</return>
}</block></function>
</unit>
```

MLSInspect: A Detection Approach For Multi-language Design Smells



Detection Process

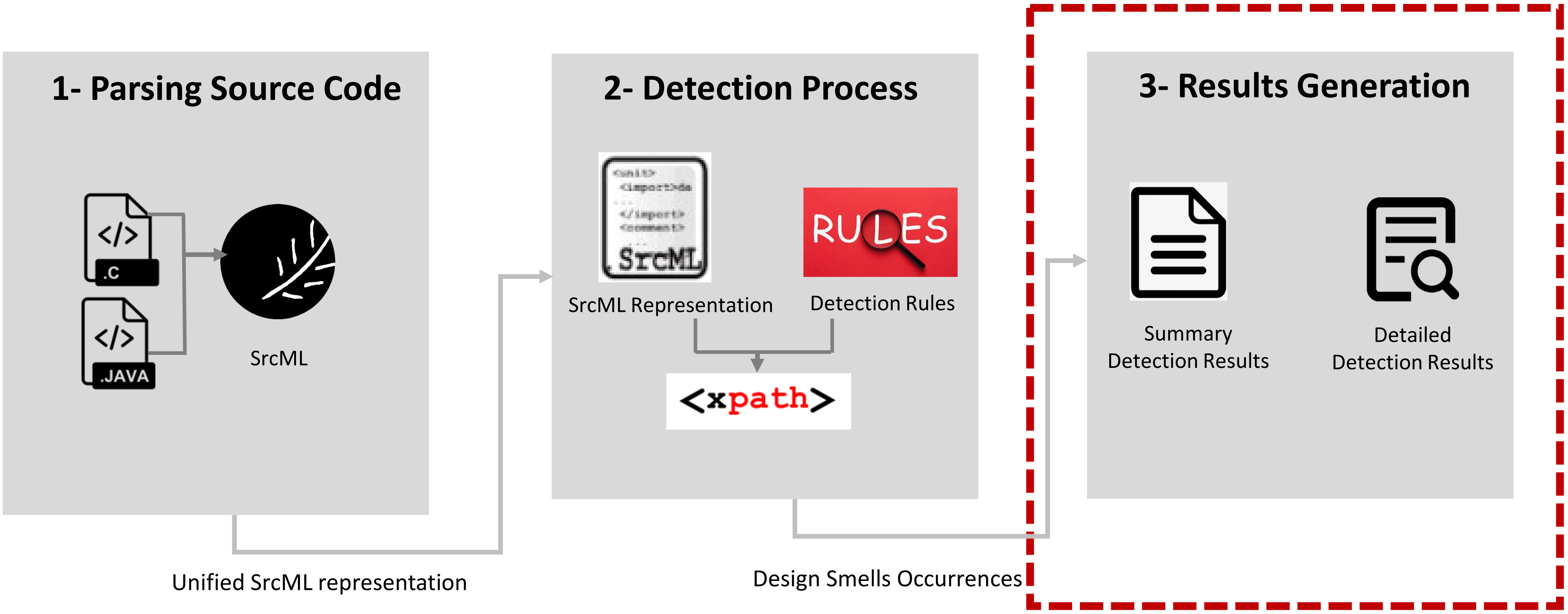
genericCallQuery = "descendant::call[name/name='%s']"

(mem ← f1(y) | f1 ∈ {GetStringChars, **GetStringUTFChars**,…})

AND (∄ f2(mem) | f2 ∈ {ReleaseGetStringChars, **ReleaseGetStringUTFChars**,…})


```
<function><type><name>JNIEXPORT</name> <name>jstring</name> <name>JNICALL</name></type> <name>Java_AccessNative_sayHello</name><parameter_list>
(<parameter><decl><type><name>JNIEnv</name> <modifier>*</modifier></type><name>env</name></decl></parameter>, <parameter><decl><type>
<name>jobject</name></type> <name>thisObj</name></decl></parameter>, <parameter><decl><type><name>jstring</name></type> <name>inJNIStr</name>
</decl></parameter>)</parameter_list> <block>{ <decl_stmt><decl><type><specifier>const</specifier> <name>char</name> <modifier>*</modifier></type>
<name>inCStr</name> <init>= <expr><call><name><operator>(</operator><operator>*</operator><name>env</name><operator>)</operator><operator>-&gt;</operator>
<name>GetStringUTFChars</name></name><argument_list>(<argument><expr><name>env</name></expr></argument>, <argument><expr><name>inJNIStr</name></expr>
</argument>, <argument><expr><name>NULL</name></expr></argument>)</argument_list></call></expr></init></decl>;</decl_stmt>
   <if>if <condition>(<expr><name>NULL</name> <operator>==</operator> <name>inCStr</name></expr>)</condition><then> <block type="pseudo"><return>return <expr>
   <name>NULL</name></expr>;</return></block></then></if> <expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">
   "In C, the received string is: %s\n"</literal></expr></argument>, <argument><expr><name>inCStr</name></expr></argument>)</argument_list></call></expr>;
   </expr_stmt> <decl_stmt><decl><type><name>char</name></type> <name><name>outCStr</name><index>[<expr><literal type="number">128</literal></expr>]</index>
   </name></decl>;</decl_stmt><expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">"Enter a String: "</literal></expr>
   </argument>)</argument_list></call></expr>;</expr_stmt><expr_stmt><expr><call><name>scanf</name><argument_list>(<argument><expr><literal type="string">"%s"
   </literal></expr></argument>, <argument><expr><name>outCStr</name></expr></argument>)</argument_list></call></expr>;</expr_stmt><return>return <expr><call>
   <name><operator>(</operator><operator>*</operator><name>env</name><operator>)</operator><operator>-&gt;</operator><name>NewStringUTF</name></name><argument_list>
   (<argument><expr><name>env</name></expr></argument>, <argument><expr><name>outCStr</name></expr></argument>)</argument_list></call></expr>;</return>
}</block></function>
</unit>
```

MLSInspect: A Detection Approach For Multi-language Design Smells



Results Generation

```
The XML of the project was created.

AssumingSafeMultiLanguageReturnValues: 12
MemoryManagementMismatch: 11
NotHandlingExceptions: 7
LocalReferencesAbuse: 0
NotCachingObjectsElements: 2
UnusedDeclaration: 16
UnusedImplementation: 0
PassingExcessiveObjects: 0
NotUsingRelativePath: 1
HardCodingLibraries: 2
UnusedParameters: 74
NotSecuringLibraries: 9
ExcessiveInterLanguageCommunication: 81
TooMuchClustering: 21
TooMuchScattering: 33
```

ID	Name	Variable	Method	Class	Package	File	File Name	System	Version	Release
CS1	UnusedDeclaration		setUseOsBuffer	EnvOptions	org.rocksdb	rocksdb-5.6.2/java/sr	EnvOptions.ja	rocksdb	rocksdb-5	8/12/2017
CS2	UnusedDeclaration		getFromBatchAndD	WriteBatchWithInde	org.rocksdb	rocksdb-5.6.2/java/sr	WriteBatchWi	rocksdb	rocksdb-5	8/12/2017
CS3	UnusedDeclaration		getFromBatch	WriteBatchWithInde	org.rocksdb	rocksdb-5.6.2/java/sr	WriteBatchWi	rocksdb	rocksdb-5	8/12/2017
CS4	UnusedDeclaration		iteratorCF	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS5	UnusedDeclaration		multiGet	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS6	UnusedDeclaration		newSstFileWriter	SstFileWriter	org.rocksdb	rocksdb-5.6.2/java/sr	SstFileWriter.	rocksdb	rocksdb-5	8/12/2017
CS7	UnusedDeclaration		getProperty0	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS8	UnusedDeclaration		compactRange0	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS9	UnusedDeclaration		useOsBuffer	EnvOptions	org.rocksdb	rocksdb-5.6.2/java/sr	EnvOptions.ja	rocksdb	rocksdb-5	8/12/2017
CS10	UnusedDeclaration		keyMayExist	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS11	UnusedDeclaration		deleteRange	WriteBatchWithInde	org.rocksdb	rocksdb-5.6.2/java/sr	WriteBatchWi	rocksdb	rocksdb-5	8/12/2017
CS12	UnusedDeclaration		openROnly	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS13	UnusedDeclaration		compactRange	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS14	UnusedDeclaration		setComparatorHan	Options	org.rocksdb	rocksdb-5.6.2/java/sr	Options.java	rocksdb	rocksdb-5	8/12/2017
CS15	UnusedDeclaration		getLongProperty	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS16	UnusedDeclaration		singleDelete	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS17	NotUsingRel	rocksdbjn	loadLibrary	RocksDB	org.rocksdb	rocksdb-5.6.2/java/sr	RocksDB.java	rocksdb	rocksdb-5	8/12/2017
CS18	HardCodingL	sharedLib	loadLibrary	NativeLibraryLoader	org.rocksdb	rocksdb-5.6.2/java/sr	NativeLibrary	rocksdb	rocksdb-5	8/12/2017
CS19	HardCodingL	jniLibraryI	loadLibrary	NativeLibraryLoader	org.rocksdb	rocksdb-5.6.2/java/sr	NativeLibrary	rocksdb	rocksdb-5	8/12/2017

	File	System	Version	Package	Release	Class	Excessive	Too much	Too much	UnusedM	UnusedM	UnusedPa	Assuming	Excessive	NotHandli	NotCachir	NotSecuri	HardCodir	NotUsingF	MemoryM	LocalRefe	FilePath
1	EnvOptions.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	EnvOptior	6	1	0	2	0	0	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ja
2	WriteBatchWithIndex.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	WriteBatc	2	1	0	3	0	0	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ja
3	RocksDB.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	RocksDB	4	1	0	9	0	0	0	0	0	0	3	0	2	0	0	rocksdb-5.6.2/ja
4	SstFileWriter.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	SstFileWri	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ja
5	Options.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	Options	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ja
6	NativeLibraryLoader.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	NativeLibi	0	0	0	0	0	0	0	0	0	0	4	2	0	0	0	rocksdb-5.6.2/ja
7	internal_stats.cc	rocksdb	rocksdb-5	rocksdb	8/12/2017		0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/dł
8	db_compaction_filter_test.	rocksdb	rocksdb-5	rocksdb	8/12/2017	KeepFilte	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/dł
9	document_db.cc	rocksdb	rocksdb-5	rocksdb	8/12/2017	Document	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ut
10	transaction_impl.cc	rocksdb	rocksdb-5	rocksdb	8/12/2017	Handler	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ut
11	full_filter_block.cc	rocksdb	rocksdb-5	rocksdb	8/12/2017		0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ta
12	StatsCallbackMock.java	rocksdb	rocksdb-5	org.rocksdb	8/12/2017	StatsCallb	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	rocksdb-5.6.2/ja

MLSInspect Evaluation

MLS Inspect

Evaluated on 6 open source projects

Systems	Recall	Precision
Openj9	93%	96%
Rocksdb	87%	95%
Conscrypt	80%	95%
PlJava	90%	99%
JNA	74%	88%
JMonkey	92%	94%

(H1) Design Smells Exist in Multi-language Systems

Catalog of Multi-language Design smells

N.	Multi-language Design Smells
1	Not Handling Exceptions
2	Not Securing Libraries
3	Local Reference Abuse
4	Memory Management Mismatch
5	Excessive Objects
6	Too Much Clustering
7	Unused Method Implementation
8	Unused Parameters
9	Assuming Safe Return Values
10	Not Using Relative Path
11	Hard Coding Libraries
12	Not Caching Objects
13	Too Much Scattering
14	Excessive Inter-language Communication
15	Unused Method Declaration

✓H1

Detection Approach

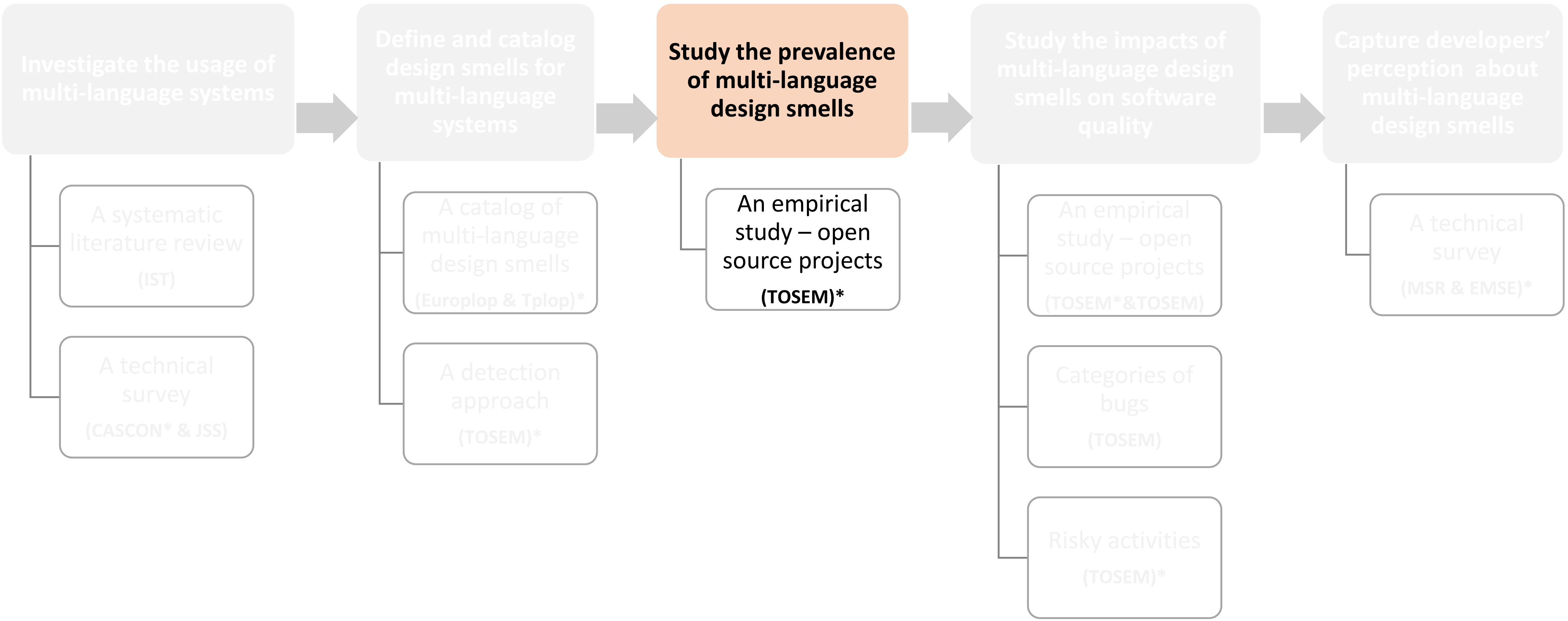

MLS Inspect

Evaluated on 6 open source projects

Minimum precision of 88%

Minimum recall of 74%

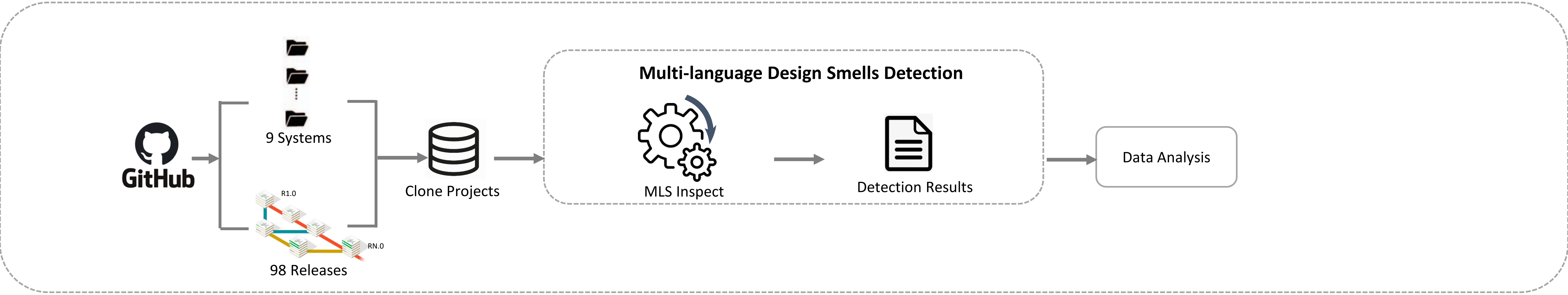
Thesis Overview



* Accepted papers

Prevalence of Multi-language Design Smells

Study Design



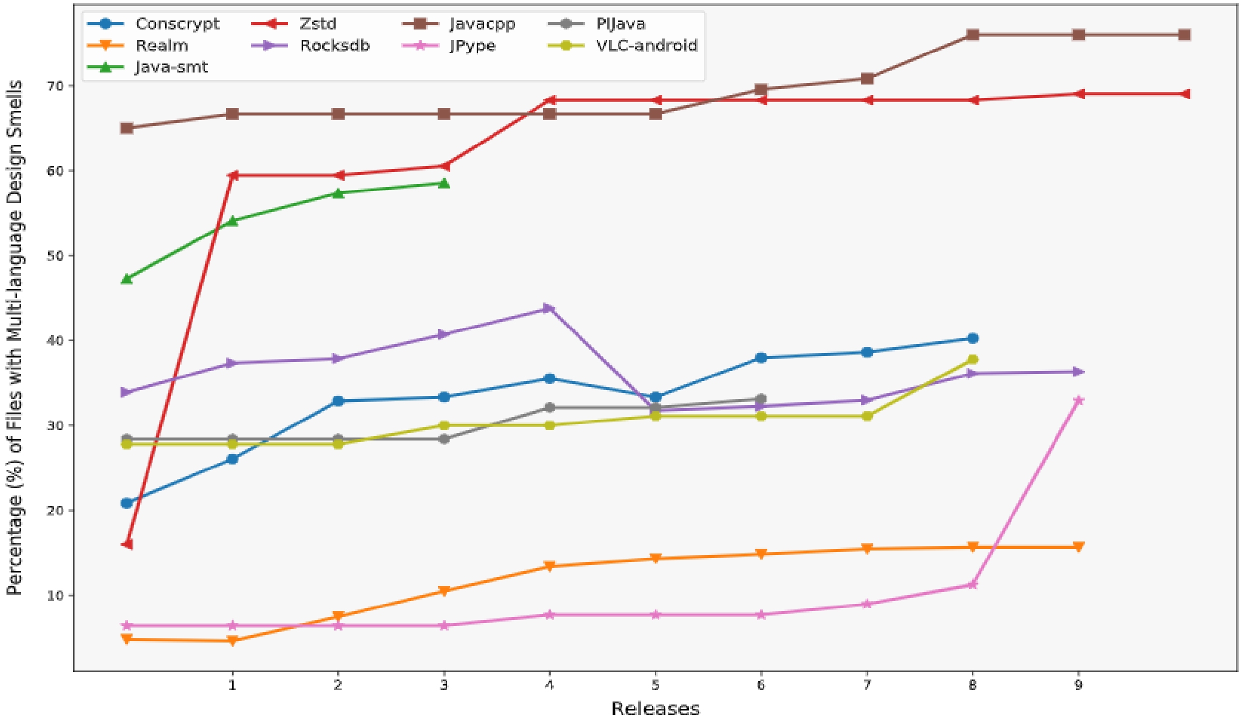
Do Multi-language Design Smells Occur Frequently in Open Source Projects?

Systems	Releases Analyzed	%Files with Smells
Conscrypt	1.0.0.RC2 - 2.3.0	30.21%
Realm	0.90.0 - 5.15.0	11.67%
Java-smt	1.0.1 - 3.0.0	36.21%
Zstd-jni	0.4.4 - latest release	61.36%
Rocksdb	5.0.2 - latest release	36.30%
Javacpp	0.9 - 1.5.1-1	58.97%
JPype	0.5.4.5 - latest release	10.18%
PlJava	REL1_5_STABLE - latest release	30.13%
VLC-android	3.0.0 – latest release	30.49%

Do Multi-language Design Smells Occur Frequently in Open Source Projects?

- Multi-language design smells are **prevalent** in open source projects

- Multi-language design smells **persist** and even **increase** over the releases



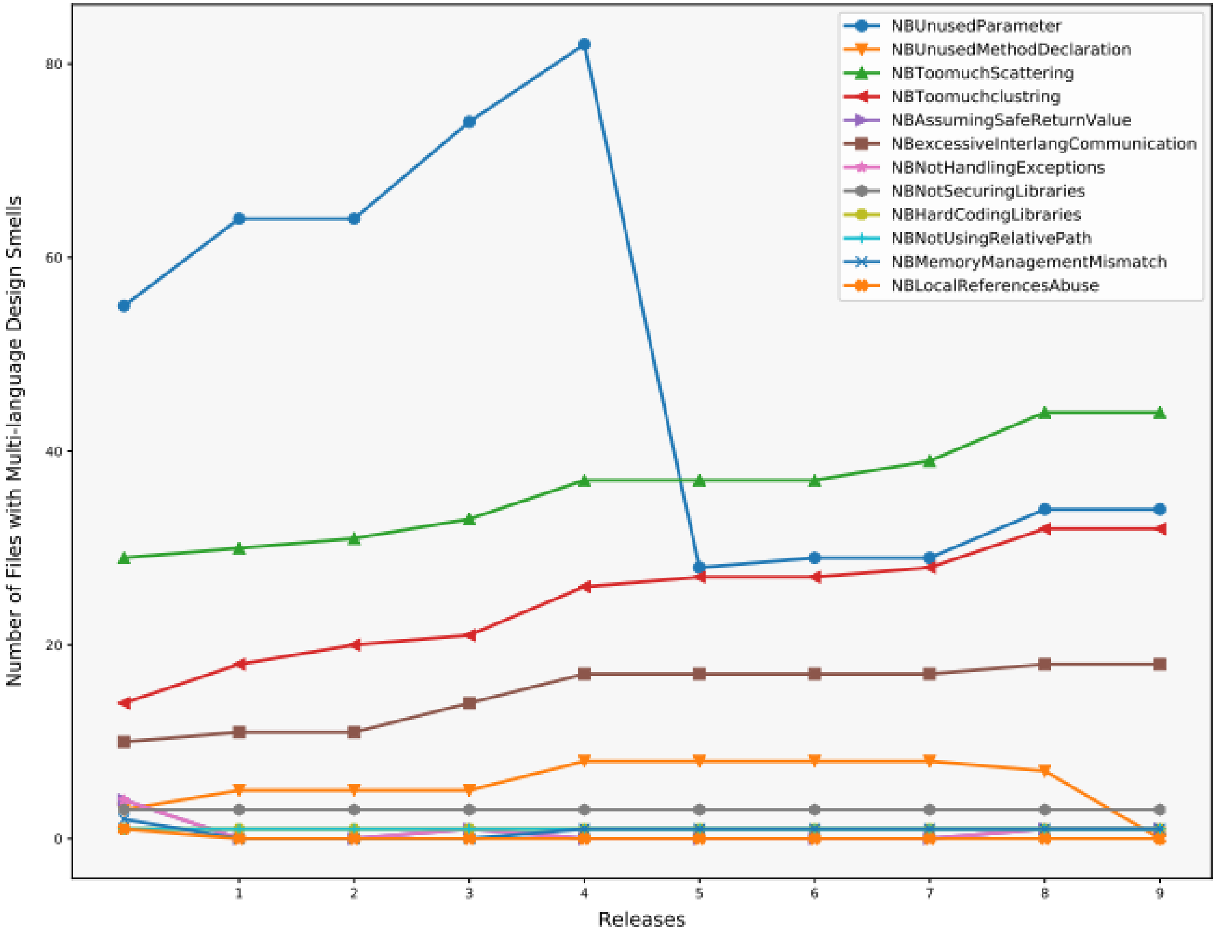
Evolution of Design Smells in the Releases of the Studied Systems

Are Some Specific Multi-language Design Smells more Frequent than Others in Open Source Projects?

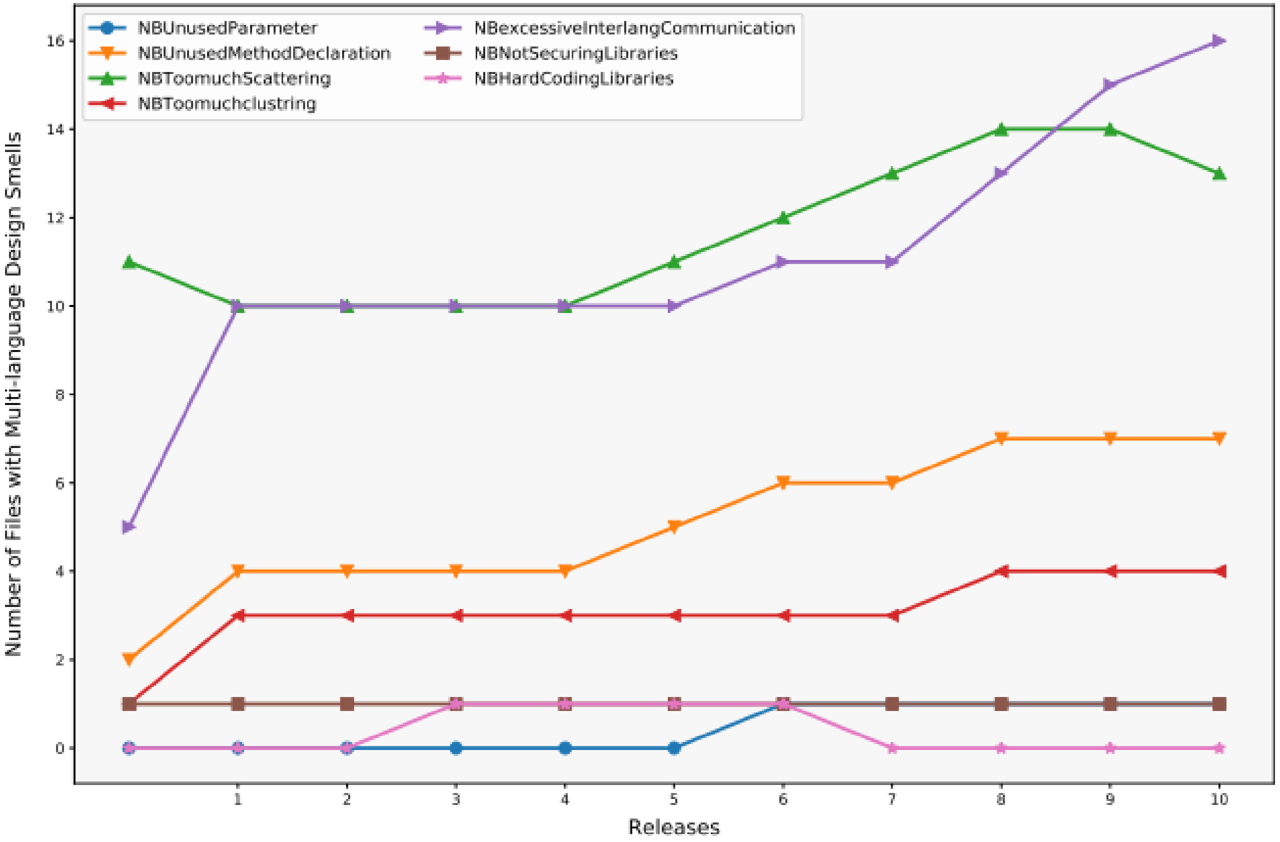
Systems	UP	UM	TMS	TMC	UMI	ASR	EO	EILC	NHE	NCO	NSL	HCD	NURP	MMM	LRA
Conscrypt	79.60%	4.40%	0%	1.90%	0%	3.99%	0%	1.90%	3.99%	0%	5.71%	0%	3.80%	3.78%	3.78
Realm	67.68%	3.066%	9.75%	14.86%	2.32%	4.33%	0%	12.58%	5.15%	0%	2.17%	0%	0 %	0%	0.79
Java-smt	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	94.06%	2.96%	2.96%	0%	0 %
Zstd	10.46%	0.95%	13.98%	12.36%	3.47%	17.98%	0%	23.55%	21.45%	0%	5.74%	3.47%	0%	2.25%	0%
Rocksdb	44.55%	5.48%	34.48%	23.47%	0%	0.67%	0%	14.35%	0.67%	0.91%	2.85%	0.95%	0.95%	0.79%	0.10%
Javacpp	2.53%	31.70%	74.19%	19.49%	0%	0%	0%	69.14%	0%	0%	6.48%	2.51%	0%	0%	0%
JPype	89.24%	0%	0%	0%	0%	1.78%	0%	0.35%	1.78%	0%	0%	0%	0%	8.25%	1.07
PlJava	64.45%	35.62%	31.02%	8.42%	2.04%	0%	0%	4.36%	2.04%	0%	0%	0%	0%	2.04%	0%
VLC-android	63.67%	25.71%	24.74%	17.10%	7.34%	3.67%	0.82%	13.29%	3.67%	0%	3.92%	0%	6.01%	0%	3.67%

Acronyms: **Up**: UnusedParameters, **UM**: UnusedMethodDeclaration, **TMS**: ToomuchScattering, **TMC**: Toomuchclustring, **UMI**: UnusedMethodImplementation , **ASR**: AssumingSafeReturnValue, **EO**: ExcessiveObjects, **EILC**: excessiveInterlangCommunication, **NHE**: NotHandlingExceptions, **NCO**: NotCachingObjects **NSL**: NotSecuringLibraries, **HCD**: HardCodingLibraries, **NURP**: NotUsingRelativePath, **MMM**: MemoryManagementMismatch, **LRA**: LocalReferencesAbuse

Evolution of Multi-Language Design Smells Over the Releases



Rocksdb



JavaCpp

(H2) Multi-language Design Smells are Prevalent

✓H2

Some Multi-language smells are more prevalent than the others:

- Unused Parameters
- Too Much Scattering
- Not Securing Library
- Excessive Inter-language Communication
- Unused Method Declaration

While others are less prevalent:

- Excessive Objects
- Not Caching Objects

Most of those smells remain and mostly increase from one release to another

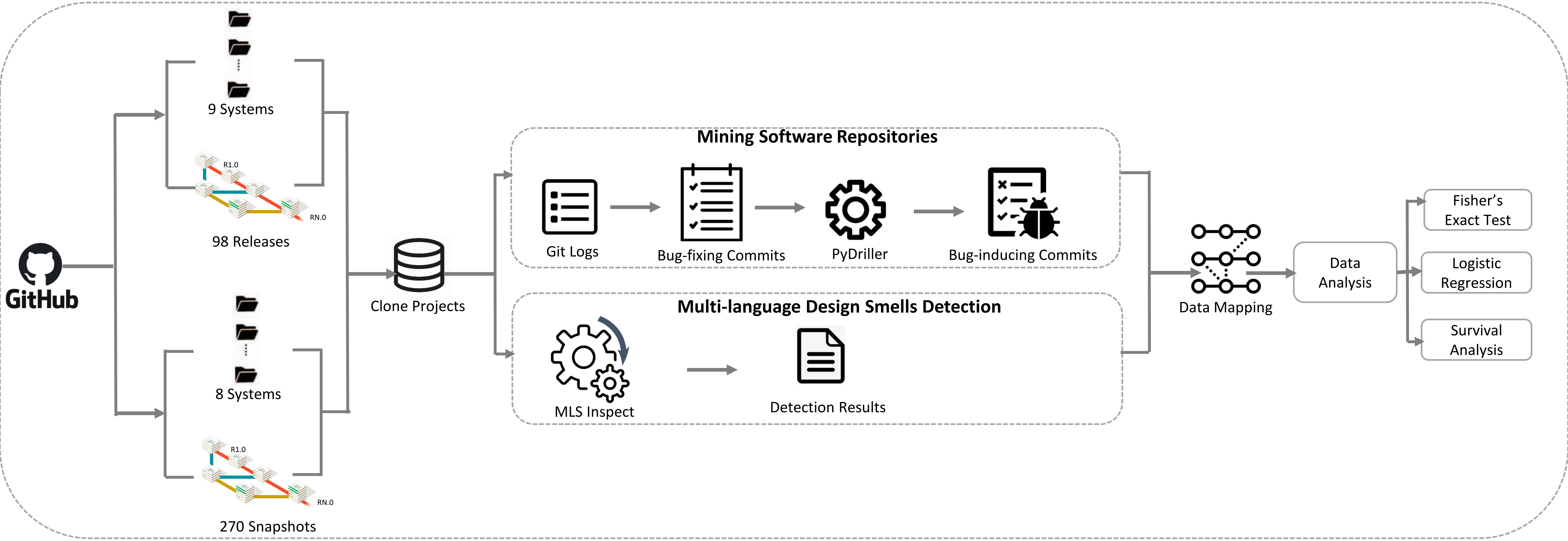
Thesis Overview

- Investigate the usage of multi-language systems
  - A systematic literature review (IST)
  - A technical survey (CASCON* & JSS)
- Define and catalog design smells for multi-language systems
  - A catalog of multi-language design smells (Europlop & Tplop)*
  - A detection approach (TOSEM)*
- Study the prevalence of multi-language design smells
  - An empirical study – open source projects (TOSEM)*
- **Study the impacts of multi-language design smells on software quality**
  - An empirical study – open source projects **(TOSEM*&TOSEM)**
  - Categories of bugs (TOSEM)
  - Risky activities (TOSEM)*
- Capture developers' perception about multi-language design smells
  - A technical survey (MSR & EMSE)*

* Accepted papers

Impacts of Multi-language Design Smells on Software Quality

Study Design



Are Files with Multi-language Design Smells more Fault-prone than Files without?

Method: Fisher's Exact Test

⚠ **Findings:** Files with occurrences of design smells can often **lead to bugs** more than files without these smells

Releases	Smelly-buggy	Buggy-NotSmelly	Smelly-NotBuggy	NotBuggy-NotSmelly	Odds ratio	p-values	Confidence Interval
rocksdb-5.0.2	82	85	17	108	6.13	<0.01	(1.2184, 2.4076)
rocksdb-5.6.2	90	80	24	107	5.01	<0.01	(1.0771, 2.1480)
pljava-1_5_0b3	32	33	14	83	5.75	<0.01	(1.0026, 2.4954)
pljava-1_5_1b2	39	36	14	76	5.88	<0.01	(1.0436, 2.4998)
pljava-1_5_2	38	34	15	78	5.81	<0.01	(1.0392, 2.4806)
realm-java-0.90.0	21	89	2	365	43.06	<0.01	(2.2938, 5.2315)
realm-java-1.2.0	20	169	2	285	16.86	<0.01	(1.3592, 4.2912)
realm-java-2.3.2	33	177	3	269	16.72	<0.01	(1.6194, 4.0135)
realm-java-3.7.2	43	165	8	271	8.82	<0.01	1.3988, 2.9570)
zstd-jni-1.3.4-1	20	1	8	12	30	<0.01	(1.2025, 5.5998
zstd-jni-latest release	22	1	7	12	37.71	<0.01	(1.4198, 5.8403)
conscrypt-1.0.0.RC2	23	20	6	90	17.25	<0.01	(1.8270, 3.8686)

Are Some Specific Multi-language Design Smells more Fault-prone than Others?

Method: Logistic Regression

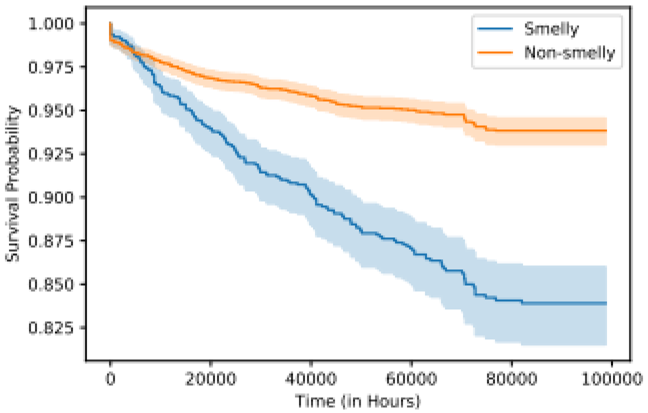
Findings: Some smells are more related to bugs than others:
- Unused Parameters
- Too Much Clustering
- Too Much Scattering
- Hard Coding Libraries
- Memory Management Mismatch

Multi-language Design Smells	Number and Percentage of Systems		
	LO > 0	LO in Top 5	(LO>0 and p<0.01)
Excessive Inter-language Communication	25%(2/8)	2	0
Too Much Clustering	62.5%(5/8)	5	4
Too Much Scattering	100%(6/6)	6	3
Unused Method Declaration	37.5%(3/8)	2	1
Unused Method Implementation	25%(1/4)	1	1
Unused Parameters	66.6%(6/9)	5	4
Not Handling Exceptions	42.8%(3/7)	3	2
Not Securing Libraries	28.5%(2/7)	2	1
Hard Coding Libraries	75%(3/4)	3	2
Memory Management Mismatch	50%(2/4)	1	1
Local References Abuse	0%(0/5)	0	0
Excessive Objects	NA	NA	NA
Not Caching Objects	NA	NA	NA

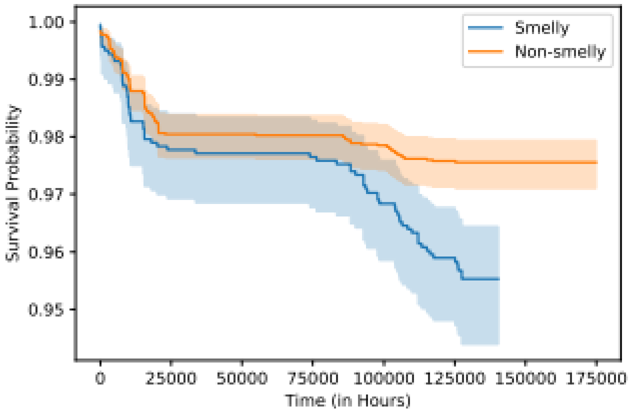
LO = Log Odds (regression coefficient estimate) of the corresponding smell from the logistic regression model.
NA = Corresponding Log odds are not available from the LR models due to singularities

Is the Risk of Bugs Higher in Files With Multi-Language Smells in Comparison With Those Without Smells?

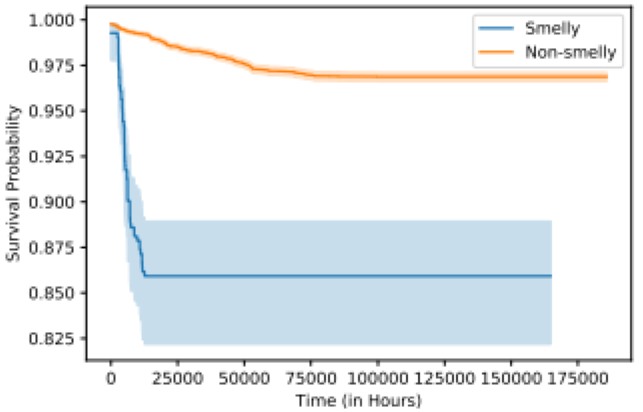
Method: Survival Analysis



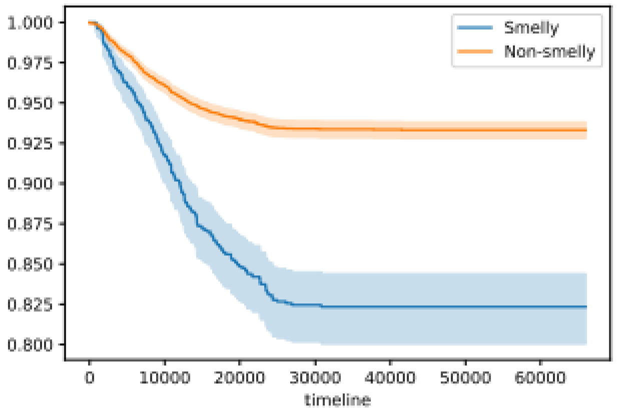
Conscrypt



PlJava



JNA

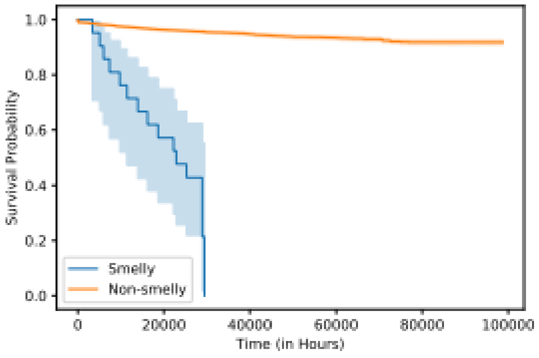


Realm

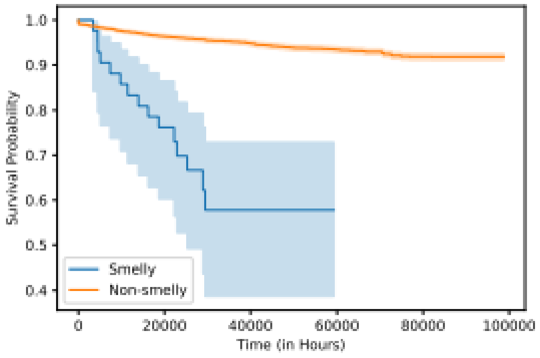
Systems	exp(coef)	p-value (CHM)	p-value (PHA)
Rocksdb	1.64	6.162e-26	1.258e-05
Frostwire	3.123	1.749e-52	0.641
Realm	2.747	7.487e-37	9.112e-05
Conscrypt	2.598	3.218e-23	0.0001
Pljava	1.805	6.425e-05	0.002
Javacpp	2.237	3.003e-08	0.164
JNA	5.033	9.526e-32	1.254e-14
OpenDDS	0.229	1.468e-09	0.992

CHM: Cox Hazard Model, **PHA**: Proportional Hazards Assumption
exp(coef): The exponentiated coefficients for the hazard ratios

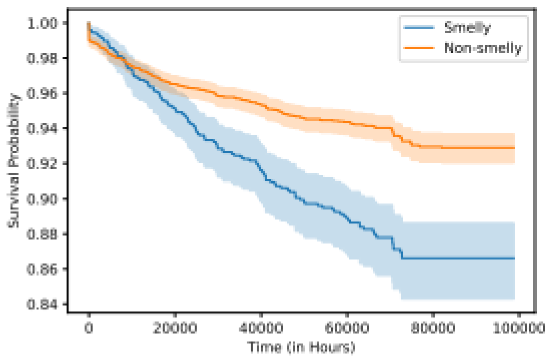
Is the Risk of Bugs Equal from One Multi-language Design Smell Type to The Other?



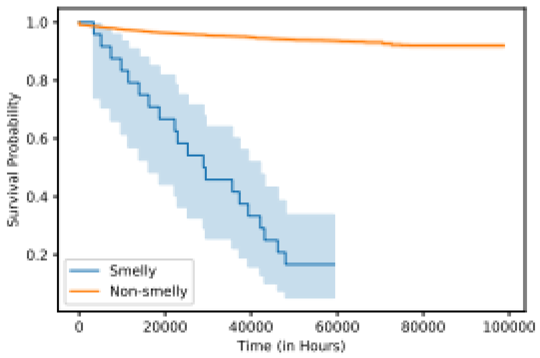
Conscrypt – Memory Management Mismatch



Conscrypt – Not Handling Exceptions



Conscrypt – Unused Parameters



Conscrypt – Local Reference Abuse

Is the Risk of Bugs Equal from One Multi-language Design Smell Type to The Other?

Method: Survival Analysis

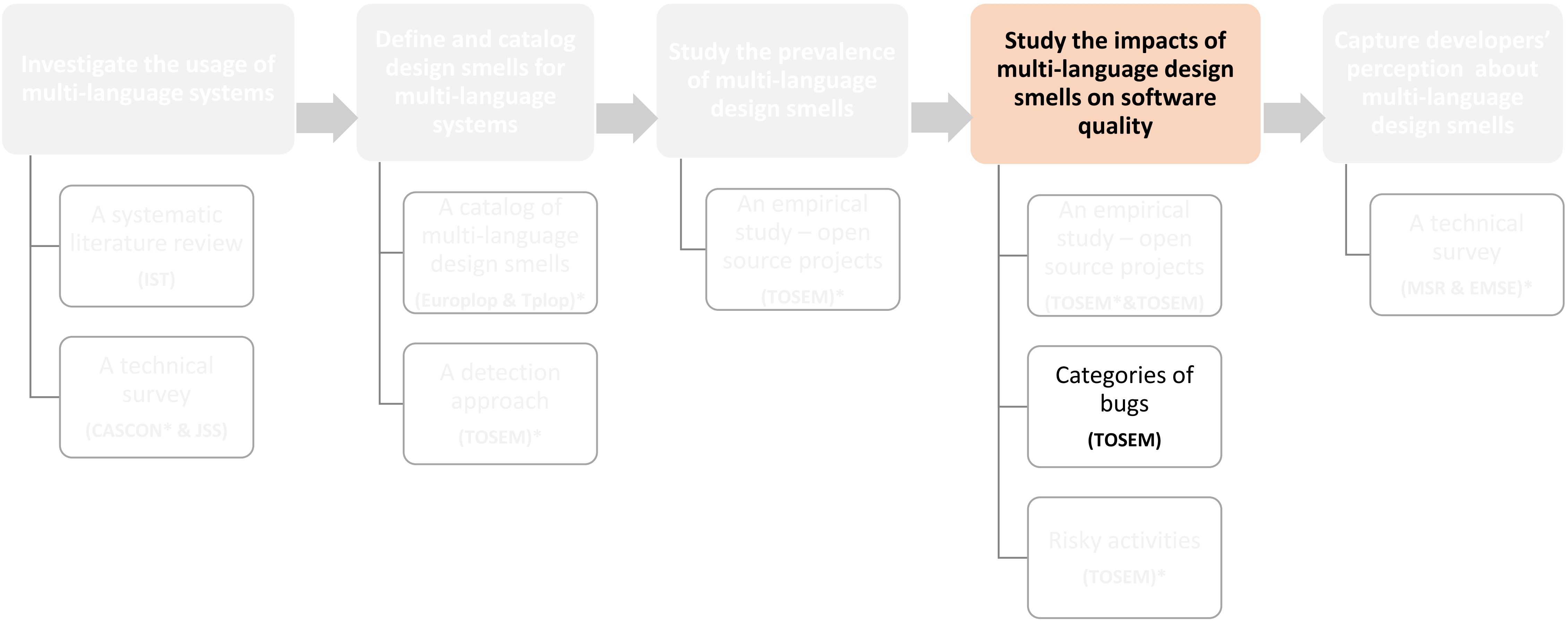
⚠ **Findings: Some smells lead faster to faults than others:**

- Memory Management Mismatch
- Hard Coding Libraries
- Unused Parameters
- Not Handling Exception
- Local Reference Abuse
- Unused Implementation

Multi-language Design Smells	#System	SFB	NSFB	% SFB	% NSFB
Unused Parameters	8	7	1	87.50%	12.50%
Unused Method Declaration	8	5	3	62.50%	37.50%
Too Much Scattering	6	3	3	50.0%	50.0%
Too Much Clustering	8	5	3	62.50%	37.50%
Unused Method Implementation	5	4	1	80.0%	20.0%
Assuming Safe Return Value	6	4	2	66.67%	33.33%
Excessive Objects	0	N/A	N/A	N/A	N/A
Excessive Interlanguage Communication	7	5	2	71.43%	28.57%
Not Handling Exceptions	7	6	1	85.71%	14.29%
Not Caching Objects	0	N/A	N/A	N/A	N/A
Not Securing Libraries	8	6	2	75.0%	25.0%
Hard Coding Libraries	2	2	0	100.0%	0.0%
Not Using Relative Path	6	3	3	50.0%	50.0%
Memory Management Mismatch	5	5	0	100.0%	0.0%
Local References Abuse	6	5	1	83.33%	16.67%

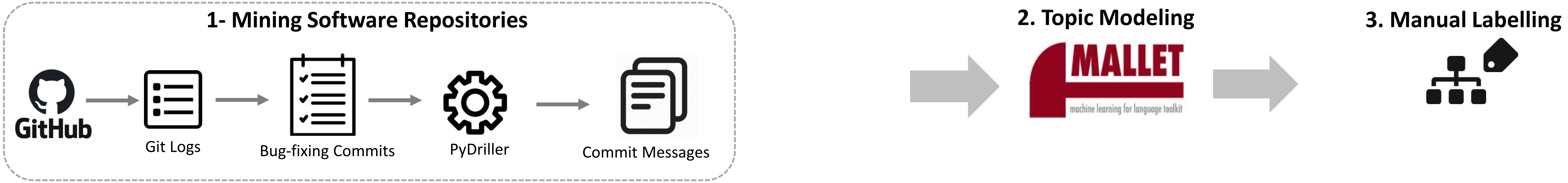
SFB: %Systems where smelly files are more bug-prone than non-smelly files
NSFB: %Systems where files without (specific) smells are more bug-prone than smelly files
#System: No. of Systems where we have hazard ratios for the concerned smell (covariate)
* Colored percentage values indicate the top-6 bug-prone smell types

Thesis Overview



* Accepted papers

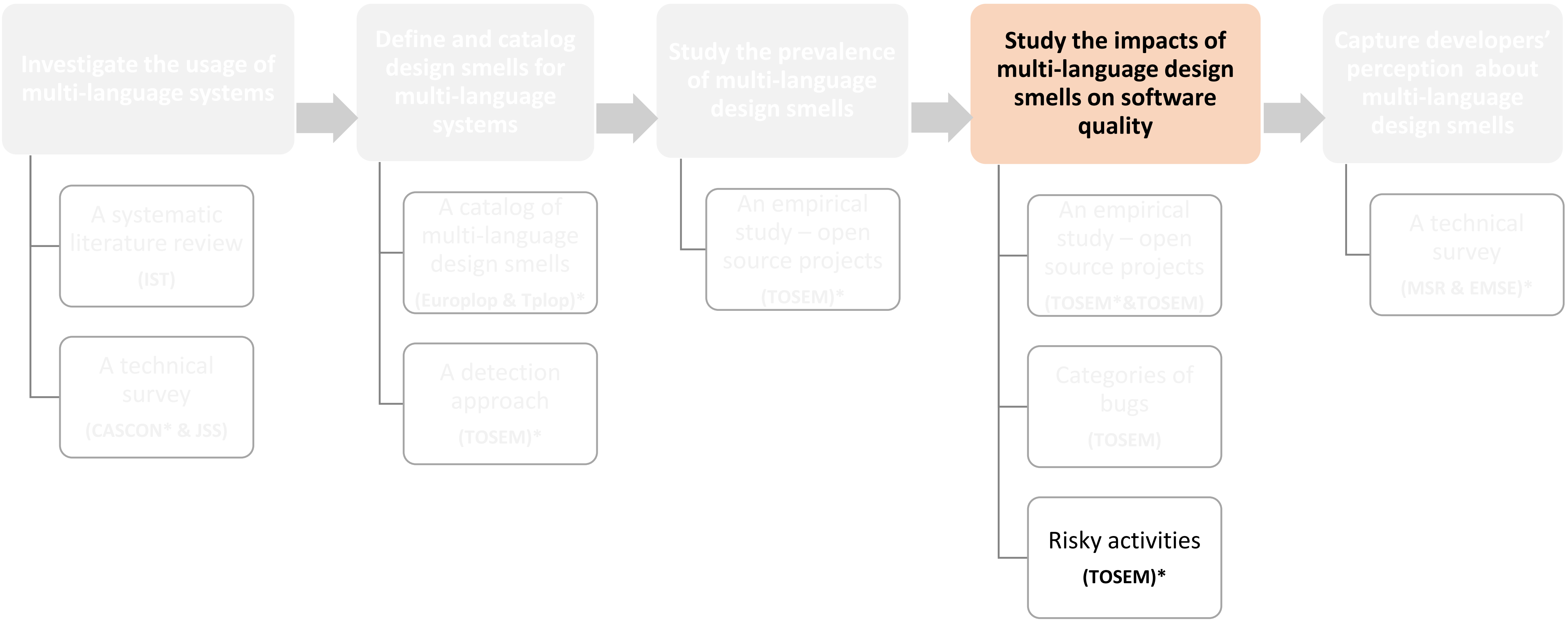
What are the Categories of Bugs that Exist in Multi-language Smelly Files?



Categories of bugs:

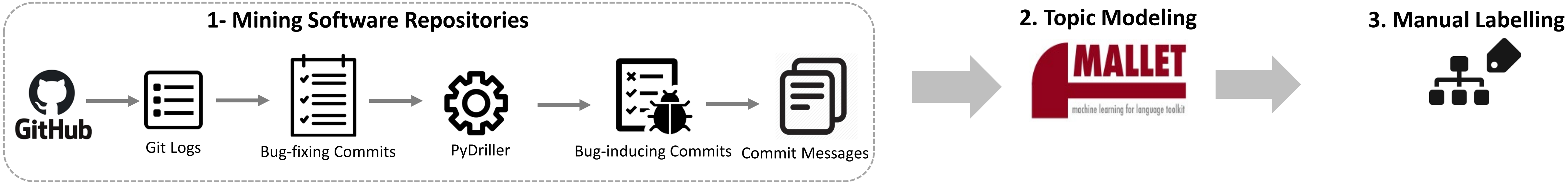
- Programming errors
- Libraries and Features Support
- Memory
- Communication and Network
- Concurrency
- Plateform and Dependencies

Thesis Overview



* Accepted papers

What are the Activities that are more Likely to Introduce Bugs in Smelly Files?



Risky Activities:

- Data conversion
- Memory management
- Exception management
- Restructuring the code
- API usage

(H3) Multi-language Design Smells Present Negative Impacts on the Software Quality

✓H3

Relationship between Smells and Bugs

⚠ **Some smells are more related to faults than others:**
- Unused Parameters
- Too Much Clustering
- Too Much Scattering
- Hard Coding Libraries
- Memory Management Mismatch

Survival Analysis

⚠ **Some smells lead faster to faults than others:**
- Memory Management Mismatch
- Hard Coding Libraries
- Unused Parameters
- Not Handling Exception
- Local Reference Abuse
- Unused Implementation

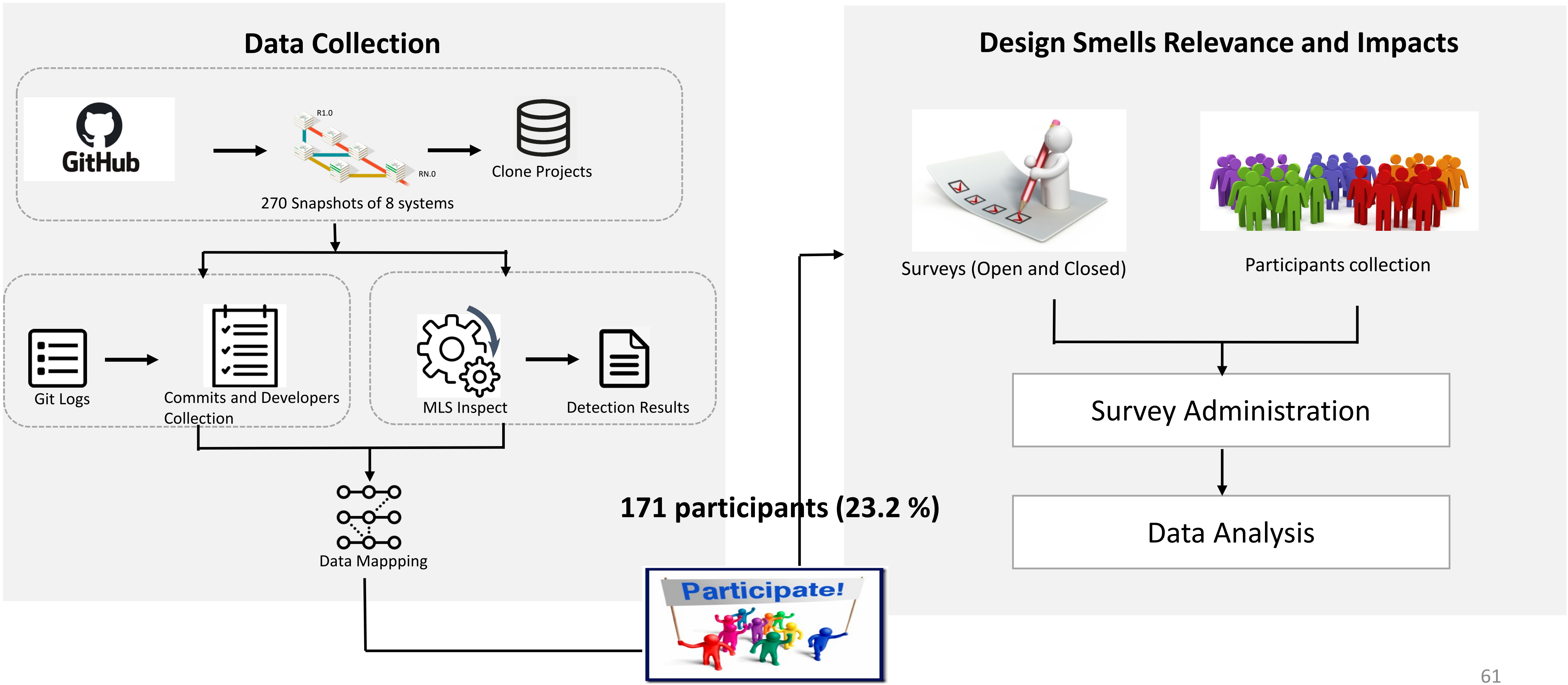
Thesis Overview

- Investigate the usage of multi-language systems
  - A systematic literature review (IST)
  - A technical survey (CASCON* & JSS)
- Define and catalog design smells for multi-language systems
  - A catalog of multi-language design smells (Europlop & Tplop)*
  - A detection approach (TOSEM)*
- Study the prevalence of multi-language design smells
  - An empirical study – open source projects (TOSEM)*
- Study the impacts of multi-language design smells on software quality
  - An empirical study – open source projects (TOSEM*&TOSEM)
  - Categories of bugs (TOSEM)
  - Risky activities (TOSEM)*
- **Capture developers' perception about multi-language design smells**
  - A technical survey **(MSR & EMSE)***

* Accepted papers

Developers' Perception about Multi-language Design Smells

Study Design



To What Extent Do Multi-language Design Smells Reflect Developers' Perception of Design Problems?

- **Most frequently identified design smells:**

  - Unused Method Implementation
  - Unused Declaration
  - Not Securing Libraries
  - Memory Management Mismatch
  - Not Caching Objects

- **Less frequently identified design smells:**

  - Hard Coding Libraries
  - Excessive Objects
  - Not Using Relative Path

Multi-language Design Smells	% of Correct Identified	% Incorrect Identified
Not Handling Exceptions	74.95%	25.05%
Not Securing Libraries	82.5%	17.5%
Local Reference Abuse	74.8%	25.2%
Memory Management Mismatch	81.9%	18.1%
Excessive Objects	38.6%	61.4%
Too Much Clustering	74.95%	25.05%
Unused Method Implementation	87.95%	12.05%
Unused Parameters	75.95%	24.05%
Assuming Safe Return Values	73.55%	26.45%
Not Using Relative Path	49.65%	50.35%
Hard Coding Libraries	31.9%	68.1%
Not Caching Objects	34.8%	65.2%
Too Much Scattering	72%	28%
Excessive Interlanguage Communication	66.75%	33.25%
Unused Method Declaration	84.3%	15.7%

What are the Design Smells that Developers Perceive as the Most Harmful?

- **Most harmful design smells:**

  - Not Handling Exception
  - Assuming Safe Return Values
  - Local Reference Abuse
  - Memory Management Mismatch
  - Excessive Inter-language Communication
  - Too Much Clustering

- **Less harmful design smells:**

  - Unused Parameters
  - Unused Method Declaration
  - Not Using Relative Path
  - Hard Coding Libraries

Multi-language Design Smells	Score (Borda Count)	Median Severity
Not Handling Exceptions	2261	12
Assuming Safe Return Value	2137	12
Local Reference Abuse	2063	11
Memory Management Mismatch	2052	9
Excessive Interlanguage Communication	2040	11
Too Much Clustering	1876	10
Not Securing Libraries	1358	7
Too Much Scattering	1342	7
Excessive Objects	1211	6
Unused Method Implementation	964	5
Not Caching Objects	812	6
Hard Coding Libraries	764	5
Not Using Relative Path	632	5
Unused Method Declaration	588	5
Unused Parameters	438	5

What are the Perceived Impacts of Multi-language Design Smells on Software Quality?

Multi-language Design Smells	Expandability	Simplicity	Reusability	Learnability	Understandability	Modularity
Not Handling Exceptions	-	-	-	-	-	-
Not Securing Libraries	-	-	-	-	-	-
Local Reference Abuse	-	-	-	-	-	-
Memory Management Mismatch	-	-	-	-	-	-
Excessive Objects	-	-	-	-	-	-
Too Much Clustering	-	-	-	-	-	-
Unused Method Implementation	-	-	-	-	-	-
Unused Parameters	-	-	-	-	-	-
Assuming Safe Return Values	-	-	-	-	-	-
Not Using Relative Path	NEU	NEU	-	NEU	-	-
Hard Coding Libraries	-	-	-	-	-	-
Not Caching Objects	-	-	-	-	-	-
Too Much Scattering	-	-	-	-	-	-
Excessive Inter-language Communication	-	-	-	-	-	-
Unused Method Declaration	-	-	-	-	-	-

- : Negative impact NEU **: Neutral Impact** **Most impacted**

What are the Perceived Impacts of Multi-language Design Smells on Software Quality?

- **Main negatively impacted quality attributes:**
  - Understandability
  - Reusability
  - Expandability

- **Less negatively impacted quality attributes:**
  - Learnability
  - Modularity

Do Developers Plan to Refactor Multi-language Design Smells?

✓ • **Design smells considered for refactoring:**

- Memory Management Mismatch
- Too Much Clustering
- Assuming Safe Return Values
- Not Securing Libraries
- Too Much Scattering

⚠ • **Design smells not considered for refactoring:**

- Excessive Objects
- Unused Method Declaration
- Unused Method Implementation

Multi-language Design Smells	%No Refactoring	% Yes Given Solution	% Yes Alternative Solution
Not Handling Exceptions	29.4	64.95%	5.65%
Not Securing Libraries	25.25	72.8%	1.95%
Local Reference Abuse	29.65	60.35%	9.9%
Memory Management Mismatch	10.9	81.45%	7.65%
Excessive Objects	62.9	31.4%	5.7%
Too Much Clustering	14.3	78.1%	7.6%
Unused Method Implementation	55.15	42%	2.85%
Unused Parameters	36.5	57.5%	5.95%
Assuming Safe Return Values	24.05	73.6%	2.35%
Not Using Relative Path	35.9	14.75%	49.3%
Hard Coding Libraries	12.5	35.4%	52.1%
Not Caching Objects	39.6	52.1%	8.3%
Too Much Scattering	23.85	66.15%	9.95%
Excessive Interlanguage Communication	49.1	15.2%	35.65%
Unused Method Declaration	55.95	41.65%	2.4%

Developers’ Perception Versus Empirical Findings (Prevalence)

Empirical investigation

- **Most prevalent design smells:**

- Unused Parameters
- Too Much Scattering
- Not Securing Libraries
- Excessive Inter-language Communication
- Unused Method Declaration

- **While others are less prevalent:**

- Excessive Objects
- Not Caching Objects

Survey

- **Frequently identified design smells:**

- Unused Parameters
- Too Much Scattering
- Not Securing Libraries
- Excessive Inter-language Communication
- Unused Method Declaration
- Not Caching Objects

- **Less frequently identified design smells:**

- Excessive Objects

Developers' Perception Versus Empirical Findings (Impact)

Empirical investigation

Some smells lead faster to bugs than others:

- Memory Management Mismatch
- Not Handling Exception
- Local Reference Abuse
- Unused Implementation
- Unused Parameters
- Hard Coding Libraries

Some smells are more related to bugs than others:

- Memory Management Mismatch
- Too Much Clustering
- Too Much Scattering
- Unused Parameters
- Hard Coding Libraries

Developers' Survey

- **Perceived as harmful design smells:**

- Memory Management Mismatch
- Not Handling Exception
- Local Reference Abuse
- Unused Implementation
- Too Much Clustering
- Too Much Scattering

- **Perceived as less harmful design smells:**

- Unused Parameters
- Hard Coding Libraries

Recommendations for Researchers

- Investigate **design smells** and **design patterns** for multi-language software development
- Investigate **why and how some specific types of smells are more frequent than others**
- Explore the **causes** and **circumstances** under which the studied **smells** may increase the **risk of bugs**
- Investigate the **roots causes** and **recommend mitigation strategies** related to the **categories of bugs**

Recommendations for Practitioners

- Developers should **pay attention** to the **design smells** studied in this thesis
- Apply MLSInspect to **detect occurrences of the studied design smells**
- **Prioritize** multi-language **smells types** for **maintenance** activities
- They could also leverage our results to better **prioritize** their **refactoring activities**

What is Next?

- Expand our study to other combinations of programming languages
- Investigate and document design patterns for multi-language systems
- Consider refactoring strategies for multi-language design smells
- Study the co-occurrence of multi-language design smells with traditional smells
- Study the combination of programming languages in machine learning applications:
    - Design smells and design patterns
    - Categories of bugs and issues

Conclusion

Multi-language Design Smells

- **Multi-language design smells** are defined as **poor design** and **coding decisions** when **bridging** between **different programming languages**
- Design smells include anti-patterns and code smells
- They represent **violations** of **best practices** related to the **combination of programming languages** that often indicate the presence of bigger problems

(H1) Design Smells Exist in Multi-language Systems

Catalog of Multi-language Design smells

N.	Multi-language Design Smells
1	Not Handling Exceptions
2	Not Securing Libraries
3	Local Reference Abuse
4	Memory Management Mismatch
5	Excessive Objects
6	Too Much Clustering
7	Unused Method Implementation
8	Unused Parameters
9	Assuming Safe Return Values
10	Not Using Relative Path
11	Hard Coding Libraries
12	Not Caching Objects
13	Too Much Scattering
14	Excessive Inter-language Communication
15	Unused Method Declaration

✓H1

Detection Approach

MLS Inspect

Evaluated on 6 open source projects

Minimum precision of 88%

Minimum recall of 74%

(H2) Multi-language Design Smells are Prevalent

✓H2

⚠ **Some Multi-language smells are more prevalent than the others:**
- Unused Parameters
- Too Much Scattering
- Not Securing Library
- Excessive Inter-language Communication
- Unused Method Declaration

⚠ **While others are less prevalent:**
- Excessive Objects
- Not Caching Objects

Most of those smells remain and mostly increase from one release to another

(H3) Multi-language Design Smells Present Negative Impacts on the Software Quality

✓H3

Relationship between Smells and Bugs

⚠ **Some smells are more related to faults than others:**
- Unused Parameters
- Too Much Clustering
- Too Much Scattering
- Hard Coding Libraries
- Memory Management Mismatch

Survival Analysis

⚠ **Some smells lead faster to faults than others:**
- Memory Management Mismatch
- Hard Coding Libraries
- Unused Parameters
- Not Handling Exception
- Local Reference Abuse
- Unused Implementation