

A Concern-Centric Empirical Evaluation of Multi-Language Code Smells: An LLM-Assisted Study of JNI Software Evolution

Md Shahrukh Ansari

shahrukh3810@gmail.com

University of Hyderabad <https://orcid.org/0009-0000-5270-2037>

Salman Abdul Moiz

University of Hyderabad <https://orcid.org/0000-0001-9294-0275>

Research Article

Keywords: Multi-language software, Java Native Interface (JNI), Code Smells, Software Evolution, Software Maintenance, Mining Software Repositories

Posted Date: September 7th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10926809/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: The authors declare no competing interests.

Highlights

A Concern-Centric Empirical Evaluation of Multi-Language Code Smells: An LLM-Assisted Study of JNI Software Evolution

Md Shahrukh Ansari, Salman Abdul Moiz

- A concern-centric methodology for empirically evaluating multi-language code and design smell catalogs, demonstrated through a large-scale study of JNI systems.
- An empirical taxonomy of JNI maintenance concerns extracted from 652 concern-addressing commits, together with an assessment of how well existing JNI code and design smells represent these concerns.
- Three candidate JNI-specific smell definitions—*Cross-Language Shotgun Surgery (CLSS)*, *Cross-Language Abstraction Leakage (CLAL)*, and *Wrong Responsibility Allocation (WRA)*—derived from recurring maintenance concerns that are not adequately represented by the current JNI smell catalog, together with their initial operationalizations.

A Concern-Centric Empirical Evaluation of Multi-Language Code Smells: An LLM-Assisted Study of JNI Software Evolution

Md Shahrukh Ansari^{a,*}, Salman Abdul Moiz^a

^aUniversity of Hyderabad, Prof. C. R. Rao Road, Gachibowli, Hyderabad, 500046, Telangana, India

ARTICLE INFO

Keywords:

Multi-language software
Java Native Interface (JNI)
Code Smells
Software Evolution
Software Maintenance
Mining Software Repositories

ABSTRACT

Multi-language software systems provide significant benefits by combining the strengths of different programming languages, but they also introduce maintainability challenges arising from cross-language dependencies. Existing research has proposed code and design smell catalogs to capture recurring cross-language design deficiencies, particularly for Java Native Interface (JNI)-based systems. However, it remains unknown whether these smell definitions correspond to the maintenance concerns that developers actually encounter and address during software evolution.

This paper presents the first concern-centric empirical evaluation of a multi-language smell catalog through software evolution history, using JNI as the target interoperability technology. We propose a methodology that combines repository mining, Large Language Model (LLM)-assisted commit-diff analysis, manual validation, and independent smell mapping to identify and categorize JNI maintenance concerns. Applying the proposed methodology to 8,207 JNI-related commits from the evolution history of 15 open-source JNI projects, we identified 652 manually validated concern-addressing commits belonging to eleven recurring maintenance concern families.

Our findings show that existing JNI smell definitions provide only partial coverage of developer maintenance concerns: only 36.5% of the identified concern-addressing commits map to existing JNI smells, while the remaining maintenance activities reveal important gaps in the current smell catalog. Analysis of the recurring unmapped concerns motivates three candidate JNI-specific smell definitions: *Cross-Language Shotgun Surgery (CLSS)*, *Cross-Language Abstraction Leakage (CLAL)*, and *Wrong Responsibility Allocation (WRA)*.

The results demonstrate that software evolution history provides valuable empirical evidence for evaluating and extending multi-language smell catalogs. More generally, the proposed concern-centric methodology offers a systematic framework for empirically validating existing smell definitions and discovering new code and design smells in multi-language software systems.

1. Introduction

Modern software systems increasingly combine multiple programming languages to leverage their complementary strengths, such as portability, execution efficiency, platform-specific capabilities, and access to mature software libraries Mayer, Kirsch and Le (2017); Abidi, Grichi and Khomh (2019a); Li, Meng, Li and Cai (2021). While this multi-language paradigm offers significant engineering benefits, it also introduces additional maintenance challenges arising from heterogeneous execution environments, language interoperability, distinct memory management mechanisms, incompatible type systems, and cross-language dependencies. Consequently, maintaining correctness, consistency, and evolvability across language boundaries is substantially more complex than in comparable single-language software systems.

To better understand and improve the maintainability of such systems, researchers have proposed several multi-language code and design smells that capture recurring cross-language design deficiencies Abidi, Grichi, Khomh and Guéhéneuc (2019b); Abidi, Khomh and Guéhéneuc (2019c); Abidi and Khomh (2020). Among the various interoperability technologies, the Java Native Interface (JNI) Liang

(1999) has received the greatest attention due to its widespread adoption for integrating Java applications with native C/C++ libraries. Existing studies have defined a catalog of JNI-specific code and design smells, developed automated detection techniques, and investigated their prevalence and impact on software quality. These studies have substantially advanced the understanding of maintainability issues in JNI-based software.

Despite these advances, an important question remains unanswered: **Do existing JNI smell definitions correspond to the maintenance concerns that developers actually encounter and address during software evolution?** Existing research has primarily adopted a design-centric perspective by deriving smell definitions from software design principles and identifying them through static source-code analysis. However, there is little empirical evidence demonstrating whether these smell definitions capture the problems that developers actually invest effort in resolving during maintenance. Consequently, the practical relevance and completeness of the existing JNI smell catalog remain largely unexplored.

A key challenge in answering this question is that developers rarely describe their maintenance activities using the terminology of code smells. Instead, maintenance commits typically address concrete software engineering problems, such as removing obsolete native interfaces, improving resource management, reducing unnecessary cross-language

^{**}Corresponding author

 shahrukh3810@gmail.com (M.S. Ansari); salman@uohyd.ac.in (S.A. Moiz)

Moiz)

ORCID(s):

communication, reorganizing responsibilities, or eliminating redundant JNI operations. These maintenance activities reflect the *underlying maintenance concerns* represented by code smells rather than the smell names themselves. Consequently, directly searching for smell instances or relying on commit messages mentioning smell terminology would overlook much of the maintenance effort devoted to addressing cross-language design deficiencies.

Motivated by this observation, we adopt a *concern-centric* rather than a smell-centric perspective. Instead of beginning with predefined smell definitions, we first identify the JNI-related maintenance concerns addressed by developers by mining software evolution history. The extracted maintenance concerns are subsequently mapped to existing JNI code and design smells to evaluate how well the current smell catalog represents the issues encountered during software maintenance. Furthermore, recurring concerns that cannot be adequately represented by existing smell definitions provide empirical evidence for identifying gaps in the current JNI smell catalog and motivate the proposal of candidate JNI-specific smell definitions.

To investigate these questions, we analyze the complete commit history of fifteen open-source JNI projects comprising 74,656 commits. We extract 8,207 JNI-related commit diffs, identify the maintenance concerns addressed by developers, and systematically map these concerns to existing JNI smell definitions. This concern-centric analysis enables us to empirically evaluate the practical relevance of the current JNI smell catalog while identifying recurring cross-language maintenance concerns that remain insufficiently represented.

Although this study focuses on JNI, the proposed concern-centric methodology is applicable to any multi-language ecosystem in which software evolution history can be mined and maintenance concerns can be related to existing smell definitions.

The following research questions guide this study:

1. **RQ1. What JNI-related maintenance concerns are addressed by developers during software evolution?**
This question identifies and categorizes the recurring JNI maintenance concerns addressed in software evolution.
2. **RQ2. Which extracted maintenance concerns are adequately represented by existing JNI code and design smells?**
This question investigates the extent to which the identified maintenance concerns correspond to existing JNI smell definitions and evaluates the empirical coverage of the current JNI smell catalog.
3. **RQ3. Which recurring maintenance concerns that are not adequately represented by existing JNI smell catalogs motivate candidate JNI-specific smell definitions?**
This question investigates whether recurring unmapped maintenance concerns exhibit sufficient recurrence and conceptual consistency to justify candidate JNI-specific smell definitions.

The main contributions of this paper are:

1. A concern-centric methodology for empirically evaluating multi-language code and design smell catalogs, demonstrated through a large-scale study of JNI systems.
2. An empirical taxonomy of JNI maintenance concerns extracted from 652 concern-addressing commits, together with an assessment of how well existing JNI code and design smells represent these concerns.
3. Three candidate JNI-specific smell definitions—*Cross-Language Shotgun Surgery (CLSS)*, *Cross-Language Abstraction Leakage (CLAL)*, and *Wrong Responsibility Allocation (WRA)*—derived from recurring maintenance concerns that are not adequately represented by the current JNI smell catalog, together with their initial operationalizations.

The remainder of this paper is organized as follows. Section 2 presents the related works. Section 3 describes the research design for this study. Section 4 reports the results. Section 5 defines the studied smells in detail. Section 6 discusses the results in detail. Section 7 presents the threats to the validity of the results. Section 8 concludes the work and discusses future works.

2. Related Work

This section presents the literature related to this study.

2.1. Multi-Language Code and Design Smells

The increasing adoption of multi-language software systems has introduced design challenges that cannot be adequately characterized using traditional mono-language code smells. By combining programming languages through interoperability mechanisms such as the Java Native Interface (JNI), developers benefit from improved performance and code reuse but also face additional maintenance challenges arising from cross-language dependencies and heterogeneous execution environments Abidi et al. (2019c,b); Abidi and Khomh (2020).

To address these challenges, Abidi *et al.* proposed the first catalog of multi-language anti-patterns, patterns, and code smells for JNI systems, derived from open-source projects, developer documentation, and bug reports Abidi et al. (2019c,b); Abidi and Khomh (2020). They subsequently developed automated detection techniques for fifteen JNI-specific smells and conducted large-scale empirical studies demonstrating their prevalence and their association with software fault-proneness and bug-proneness Abidi, Rahman, Openja and Khomh (2021, 2024).

Ansari *et al.* Ansari and Moiz (2024) also developed an improved automated detection technique for two of the existing smells and studied the impact of multi-language smells on fault- and change-proneness Ansari and Moiz (2025).

While these studies have substantially advanced the understanding of multi-language code and design smells,

they adopt a predominantly design-centric perspective by defining smells from design principles and detecting them through static analysis. In contrast, our work adopts a maintenance-centric perspective by investigating the JNI-related concerns that developers actually address during software evolution and evaluating the extent to which these concerns are represented by existing JNI smell definitions.

2.2. Large Language Models for Commit Analysis

Large Language Models (LLMs) have recently emerged as effective tools for a wide range of software engineering tasks, including code understanding, program analysis, software maintenance, refactoring, documentation, and software evolution. Recent surveys by Hou *et al.* Hou, Zhao, Liu, Yang, Wang, Li, Luo, Lo, Grundy and Wang (2024) and subsequent reviews Jelodar, Meymani and Razavi-Far (2025) identify software maintenance and source-code understanding as major application areas of LLMs, demonstrating their ability to extract semantic information from diverse software artifacts.

Among these applications, commit-level analysis has received considerable attention. Commit message generation requires a model to infer the semantic intent of source code modifications from commit diffs and summarize them in natural language. Zhang *et al.* Zhang, Zhao, Wang and Liang (2024) demonstrated that ChatGPT and other LLMs can effectively understand commit diffs and generate high-quality commit messages, with human evaluators frequently preferring LLM-generated summaries over existing approaches. Similarly, Xue *et al.* Xue, Wu, Yu, Jin, Yang, Li, Yang and Tan (2024) conducted a comprehensive empirical study of automated commit message generation and reported that GPT-models consistently outperform previous state-of-the-art techniques on commit understanding tasks.

These studies demonstrate that LLMs are capable of extracting semantically meaningful information from software evolution artifacts. However, existing work has primarily focused on commit summarization and commit message generation. To the best of our knowledge, no prior study has employed LLMs to identify maintenance concerns from commit diffs and subsequently evaluate their relationship with existing multi-language code and design smells. Our work addresses this gap through an LLM-assisted, human-in-the-loop methodology for mining JNI-related maintenance concerns from software repositories.

3. Study Design

The objective of this study is to investigate whether the maintenance concerns addressed by developers in JNI-based systems are adequately represented by existing JNI code and design smells and, where gaps exist, to identify recurring concerns that motivate new smell definitions. To achieve this objective, we analyze the software evolution history of open-source JNI projects through a multi-stage methodology, illustrated in Figure 1.

The study begins by constructing a dataset of open-source JNI projects satisfying the selection criteria described

in Section 3.1. The complete Git history of each repository is then analyzed to identify commits related to JNI development, from which the corresponding commit diffs are extracted. These diffs, together with their associated commit messages, constitute the primary unit of analysis throughout the study.

The extracted commit diffs are subsequently analyzed using a Large Language Model (LLM) to identify the JNI-related maintenance concerns addressed by each commit. Rather than classifying commits directly into smell categories, the LLM is instructed to identify the underlying maintenance concerns reflected by the code changes and supporting commit messages. Since LLM-generated results may contain inaccuracies, every identified concern is manually inspected and validated by the researchers. The validated concerns are then grouped into recurring concern families, forming the empirical basis for answering **RQ1**.

Next, each validated concern is systematically mapped to the existing catalog of JNI code and design smells based on the conceptual correspondence between the addressed maintenance concern and the underlying design deficiency represented by the smell. Concerns that cannot be adequately represented by any existing smell are classified as *unmapped*. This mapping process enables us to assess the coverage of existing JNI smells with respect to real-world maintenance activities, thereby addressing **RQ2**.

Finally, the recurring unmapped concern families are analyzed to determine whether they represent previously uncharacterized cross-language design deficiencies. Only concerns exhibiting sufficient recurrence and conceptual consistency are considered candidates for new JNI smell definitions. Based on this analysis, we propose three JNI-specific smell definitions together with their representative manifestations and initial operationalizations, thereby answering **RQ3**.

3.1. Data Collection

This study utilizes the same set of fifteen open-source JNI-based projects employed in our previous empirical study on the influence of multi-language design smells on software quality Ansari and Moiz (2025). While the previous study analyzed selected software releases to investigate the relationship between multi-language design smells and software quality, the present study analyzes the complete Git history of these projects to investigate the maintenance concerns addressed during software evolution.

The selected projects span diverse application domains, including database systems, multimedia frameworks, cryptographic libraries, language bridges, compression libraries, scientific computing, and communication software. Together, they represent a broad range of software sizes, development histories, and Java-native language compositions, providing a representative corpus for studying maintenance activities in JNI-based software systems.

The projects were originally selected according to the following criteria: (i) the project is publicly available and actively maintained, (ii) both Java and native C/C++ source

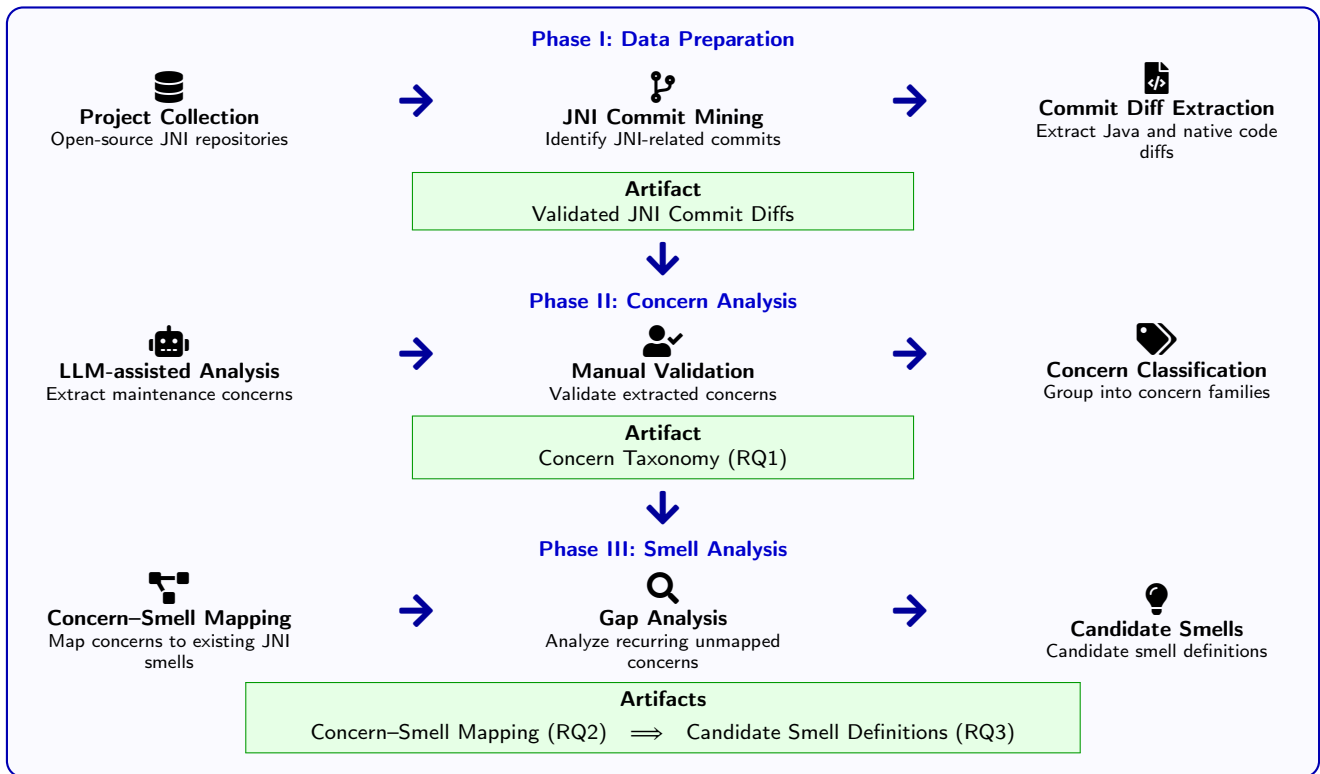


Figure 1: Overview of the proposed methodology. Phase I extracts JNI-related commit diffs from open-source repositories. Phase II employs an LLM-assisted analysis followed by manual validation to identify maintenance concerns and construct the concern taxonomy (RQ1). Phase III maps the extracted concerns to existing JNI smell definitions (RQ2) and analyzes recurring unmapped concerns to identify candidate JNI-specific smell definitions (RQ3).

Table 1
Overview of the analyzed open-source JNI projects.

Project	Domain	LOC	Java	C/C++	# Commits
RocksDB ¹	Database	487,853	11.0%	83.1%	8,375
VLC-Android ²	Multimedia	125,037	10.1%	6.7%	12,697
Realm-java ³	Mobile Database	171,705	82.0%	8.1%	8,244
Conscrypt ⁴	Cryptography	91,765	85.3%	14.0%	3,874
PL/Java ⁵	Database	71,910	67.0%	29.7%	1,236
JavaCPP ⁶	Compiler	28,713	98.0%	0.6%	658
Zstd-JNI ⁷	Compression	72,824	4.3%	92.1%	423
JPytype ⁸	Language Bridge	53,826	7.8%	58.0%	895
JavaSMT ⁹	Scientific Computing	42,049	88.0%	4.6%	1,822
JNI-Bind ¹⁰	Library	35,908	6.6%	87.7%	438
WebRTC-Java ¹¹	Communication	32,596	39.9%	55.7%	337
Monero-Java ¹²	Cryptocurrency	103,547	91.5%	8.1%	2,904
WolfSSLJNI ¹³	Communication	61,873	78.0%	20.5%	1,196
SQLite-JDBC ¹⁴	Database Engine	419,757	2.7%	80.9%	30,854
WolfCrypt-JNI ¹⁵	Cryptography	74,597	84.2%	15.0%	703
Total (15 Projects)		1,873,960			74,656

code are maintained within the same repository, (iii) Java and native components interact through the Java Native Interface (JNI), and (iv) the complete Git history is publicly available. The analyzed projects are summarized in Table 1. Detailed information regarding the project selection procedure and dataset construction can be found in our previous study Ansari and Moiz (2025). Unlike

Unlike

¹<https://github.com/facebook/rocksdb>

²<https://github.com/videlal/vlc-android>

³<https://github.com/realm/realm-java>

⁴<https://github.com/google/conscript>

⁵<https://github.com/tada/pljava>

⁶<https://github.com/bytedeco/javacpp>

⁷<https://github.com/luben/zstd-jni>

⁸<https://github.com/jpye-project/jpy>

⁹<https://github.com/sosy-lab/java-smt>

¹⁰<https://github.com/google/jni-bind>

our previous study, which analyzed selected releases, this work does not perform release-based analysis. Instead, we mine the complete commit history of each repository and analyze every commit to identify those involving JNI-related modifications. The extracted JNI-related commit diffs constitute the primary source of evidence for the subsequent maintenance concern analysis.

3.2. Extraction of JNI-Related Commit Diffs

The objective of this stage is to identify commits that potentially contain JNI-related maintenance activities.

For each repository, the complete Git history was traversed using GitPython. Every commit modifying at least one Java or native source file was inspected. Since not every modification to Java or C/C++ code is related to JNI, we further filtered commits using JNI-specific language constructs.

A source file was considered JNI-related if it contained one or more JNI constructs, including:

- native method declarations in Java,
- JNIEXPORT/JNICALL, JNIEnv, jobject, jclass, jstring, jmethodID, jfieldID, Java_* exported functions, RegisterNatives(), JNINativeMethod, inclusion of jni.h, and other JNI-specific APIs. in C/C++ files.

For every selected commit, the following information was extracted:

- commit identifier,
- commit message,
- modified Java files, and
- modified native files.

These commit diffs constitute the primary unit of analysis throughout the remainder of the study.

3.3. LLM-Assisted Maintenance Concern Extraction

The extracted commit diffs were analyzed to identify JNI-related maintenance concerns addressed during software evolution.

Given the large number of commits and the complexity of simultaneously reasoning about commit messages, Java modifications, native code modifications, and cross-language interactions, performing exhaustive manual analysis for every commit would be prohibitively time-consuming. Recent studies have shown that Large Language Models (LLMs) are effective for software engineering tasks involving source code understanding, software maintenance, program analysis, and commit-level reasoning, particularly

commit message generation, where understanding the semantic intent of code changes is essential Hou et al. (2024); Zhang et al. (2024); Jelodar et al. (2025); Xue et al. (2024).

Based on this evidence, we employ ChatGPT as an analysis assistant to identify candidate JNI-related maintenance concerns from commit diffs. ChatGPT was selected because recent empirical studies report strong performance of GPT-models on commit understanding and commit message generation tasks Xue et al. (2024). The LLM is used solely to assist the qualitative coding process by generating candidate concerns and supporting evidence. All identified concerns, their mappings to existing smells, and the proposed smell definitions are subsequently verified and finalized through manual inspection by the researchers.

The LLM received:

- the complete unified diff which contains the commit message, modified java code, and modified native code, and
- the prompt.

The analysis followed a predefined prompting protocol designed specifically for this study.

The protocol instructed the LLM to:

- analyze both the commit message and the actual code changes;
- inspect modifications in both Java and native code;
- identify only maintenance activities addressing existing JNI-related concerns;
- exclude feature additions, newly introduced functionality, code formatting changes, API evolution, and architectural restructuring that did not resolve an existing maintenance issue;
- identify all JNI-related concerns addressed by a commit rather than assigning a single label;
- report the evidence supporting each identified concern.

The prompt emphasized identifying underlying maintenance concerns rather than directly classifying commits into predefined smell categories. The complete prompt is provided in the online replication package ¹.

3.3.1. Manual Validation

Although recent studies have demonstrated the effectiveness of Large Language Models (LLMs) for software engineering tasks involving source code understanding, software maintenance, and commit-level reasoning, LLM-generated analyses may still contain incorrect or unsupported interpretations. Since manually inspecting all extracted commits is prohibitively expensive, we manually validated a representative sample comprising approximately 10% of the

¹¹<https://github.com/devopvoid/webrtc-java>

¹²<https://github.com/woodser/monero-java>

¹³<https://github.com/wolfSSL/wolfssljni>

¹⁴<https://github.com/xerial/sqlite-jdbc>

¹⁵<https://github.com/wolfSSL/wolfcrypt-jni>

¹<https://github.com/shahrulkh-uohyd/Commit-Analysis-JSS/tree/master>

identified concern-addressing commits. Specifically, out of the 652 commits identified by the LLM, 65 commits were randomly sampled for validation. Each sampled commit was independently examined by the researchers using both the commit message and the corresponding commit diff. During validation, we verified whether (i) the commit genuinely addressed one or more JNI-related maintenance concerns, and (ii) the concern(s) identified by the LLM were supported by the evidence present in the code changes and commit message.

A commit was considered a *true positive (TP)* if the manually identified concern(s) agreed with the LLM’s output. Otherwise, it was classified as a *false positive (FP)*. The precision of the LLM-assisted concern extraction was computed as:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

Among the 65 manually validated commits, 63 were confirmed as true positives, while 2 were identified as false positives, resulting in a precision of **96.92%**. This high precision indicates that the LLM-assisted extraction provides a reliable basis for the subsequent empirical analyses.

Recall was not evaluated because the objective of this study is not to exhaustively identify every commit addressing JNI-related maintenance concerns. Rather, our goal is to obtain reliable empirical evidence that developers actively address such concerns during software maintenance. Estimating recall would require constructing a complete ground-truth dataset containing every concern-addressing commit across all analyzed repositories, which is infeasible due to the absence of such a benchmark. Consequently, precision is the more appropriate evaluation metric for assessing the reliability of the LLM-assisted concern extraction in the context of this study.

3.4. Mapping Maintenance Concerns to Existing JNI Smells

Following the construction of the maintenance concern taxonomy, each identified maintenance concern was systematically mapped to the existing catalog of JNI code and design smells. The objective of this stage was to evaluate the extent to which the maintenance concerns addressed by developers are represented by existing JNI smell definitions.

The mapping was performed based on the conceptual correspondence between the underlying maintenance concern and the design deficiency represented by each smell, rather than by applying existing smell detection rules. This approach ensured that the mapping reflected the maintenance intent underlying each commit instead of the specific implementation characteristics targeted by automated detection techniques. Since a single maintenance concern may address multiple design deficiencies, a concern was allowed to map to multiple JNI smell categories whenever justified by the observed maintenance activity.

To reduce subjective bias, the mapping was conducted independently by two authors of this study. Each author assigned every identified maintenance concern to one or more existing JNI smell categories based on the published smell definitions. To assess the reliability of the mapping process, we computed Cohen’s κ based on the binary classification of each maintenance concern as either *Mapped* or *Unmapped*. The first author mapped 255 maintenance concerns to one or more existing JNI smells, while the second author mapped 248 concerns. The two authors agreed on 625 of the 652 maintenance concerns, resulting in a Cohen’s κ of 0.913, indicating *almost perfect agreement* according to the interpretation of Landis and Koch Landis and Koch (1977).

The independently produced mappings were subsequently compared. Cases in which both authors reached the same conclusion were retained as the final mappings. For the remaining cases, the authors discussed the disagreements. Whenever a consensus could not be reached, the maintenance concern was conservatively classified as *Unmapped*. This conservative strategy avoids overestimating the empirical coverage of the existing JNI smell catalog and ensures that only mappings supported by clear conceptual agreement are included in the subsequent analysis.

3.5. Identification of Candidate JNI Smell Definitions

The final stage of the methodology aims to identify candidate JNI-specific smell definitions from maintenance concerns that are not adequately represented by the existing JNI code and design smell catalog. While unmapped maintenance concerns indicate potential gaps in the current smell catalog, the absence of a corresponding smell definition does not necessarily imply the existence of a new design deficiency. Similarly, isolated or project-specific maintenance activities are insufficient for motivating new smell definitions. Therefore, only unmapped concerns exhibiting recurring and conceptually consistent maintenance patterns were considered for further analysis.

An unmapped maintenance concern was considered a candidate JNI smell if it satisfied all of the following criteria:

1. **Recurrence.** The concern was observed repeatedly across multiple commits and multiple projects, indicating that it represents a recurring maintenance issue rather than an isolated implementation decision.
2. **Conceptual Consistency.** The observed maintenance activities addressed the same underlying design or architectural problem instead of a collection of unrelated implementation issues.
3. **Lack of Representation.** The concern could not be adequately represented by any existing JNI code or design smell based on the independent manual mapping described in Section 3.4.
4. **Cross-Language Nature.** The underlying problem originated from interactions across the Java-native boundary rather than from language-specific implementation concerns.

The unmapped concern families satisfying the above criteria were subsequently analyzed to determine whether they represent distinct cross-language design deficiencies. For each candidate, we synthesized a conceptual smell definition describing the underlying design concern, identified representative manifestations observed in the analyzed repositories, formulated an initial operationalization for automated detection, and discussed possible refactoring strategies.

It is important to distinguish between a smell definition and its operationalization. A smell represents an underlying design deficiency, whereas an operationalization identifies one or more observable manifestations of that deficiency in source code. Accordingly, the operationalizations proposed in this work are derived from representative maintenance patterns observed in the analyzed repositories and should be interpreted as initial detection strategies rather than exhaustive characterizations of the corresponding smells. Additional manifestations may be incorporated as further empirical evidence becomes available.

4. Results

In this section, we present the findings from our empirical study. The results are organized around the two research questions: (RQ1) the extraction of concern taxonomy, (RQ2) mapping the concerns to existing smells, and (RQ3) proposing candidate smells for unmapped concerns.

4.1. RQ1: What JNI-related maintenance concerns are addressed by developers during software evolution?

To answer RQ1, we analyzed the complete commit history of the fifteen selected JNI projects. The analyzed repositories collectively contain 74,656 commits. Using the JNI commit extraction procedure described in Section 3.2, we identified 8,207 commits that modified JNI-related artifacts. These commits were subsequently analyzed using the proposed LLM-assisted concern extraction process, 652 commits were confirmed to explicitly address one or more JNI-related maintenance concerns. The remaining JNI-related commits primarily correspond to feature additions, functionality extensions, bug fixes unrelated to JNI maintenance concerns, refactorings without explicit concern resolution, documentation updates, build changes, and other routine development activities.

The concern-addressing commits were then categorized according to the primary maintenance concern they addressed. The resulting taxonomy consists of eleven recurring JNI maintenance concern families, representing the major categories of JNI maintenance activities observed across the analyzed repositories.

Table 2 summarizes the overall distribution of the extracted concern families, while Figure 2 illustrates their distribution across the analyzed projects.

The analysis reveals that JNI Boundary Safety is the most frequently addressed maintenance concern, accounting for 222 commits (33.80%). These commits primarily involve

Table 2

Overall distribution of extracted JNI maintenance concern families.

Concern Family	Commits
JNI Boundary Safety	222
JNI Resource Leakage	164
Others/Miscellaneous	80
Cross-Language Communication	52
JNI Thread Management	28
Unused Code Removal	28
Change Propagation Across Language	21
Cross-language Responsibility Relocation	17
Cross Language Information Hiding	16
Library Loading	14
JNI Architecture and Separation of Concerns	10
Total	652

activities that improve the correctness and robustness of interactions across the Java-native boundary, such as validating JNI usage, ensuring safe cross-language interactions, and preventing boundary-related failures. The prominence of this concern indicates that maintaining correctness at the language boundary is a primary maintenance activity in JNI systems.

The second most frequently observed concern is JNI Resource Leakage, comprising 164 commits (25.39%). These commits address resource-management issues involving JNI references, native resources, and object lifetimes, highlighting the substantial maintenance effort devoted to preventing resource leaks and ensuring proper cleanup across the language boundary.

Together, JNI Boundary Safety and JNI Resource Leakage account for approximately 59.2% of all identified concern-addressing commits, indicating that correctness and resource management dominate maintenance activities in JNI-based software.

The third largest category, Others/Miscellaneous, contains 80 commits (12.46%). These commits correspond to JNI-related maintenance activities that could not be consistently grouped into the other recurring concern families but nevertheless represent genuine developer maintenance efforts.

Beyond these dominant categories, developers also addressed several other recurring concerns:

- Cross-Language Communication: 52 commits (7.48%),
- JNI Thread Management: 28 commits (4.36%),
- Unused Code Removal: 28 commits (4.36%),
- Change Propagation Across Language: 21 commits (3.27%),
- Cross-language Responsibility Relocation: 17 commits (2.65%),
- Cross-Language Information Hiding: 16 commits (2.49%),

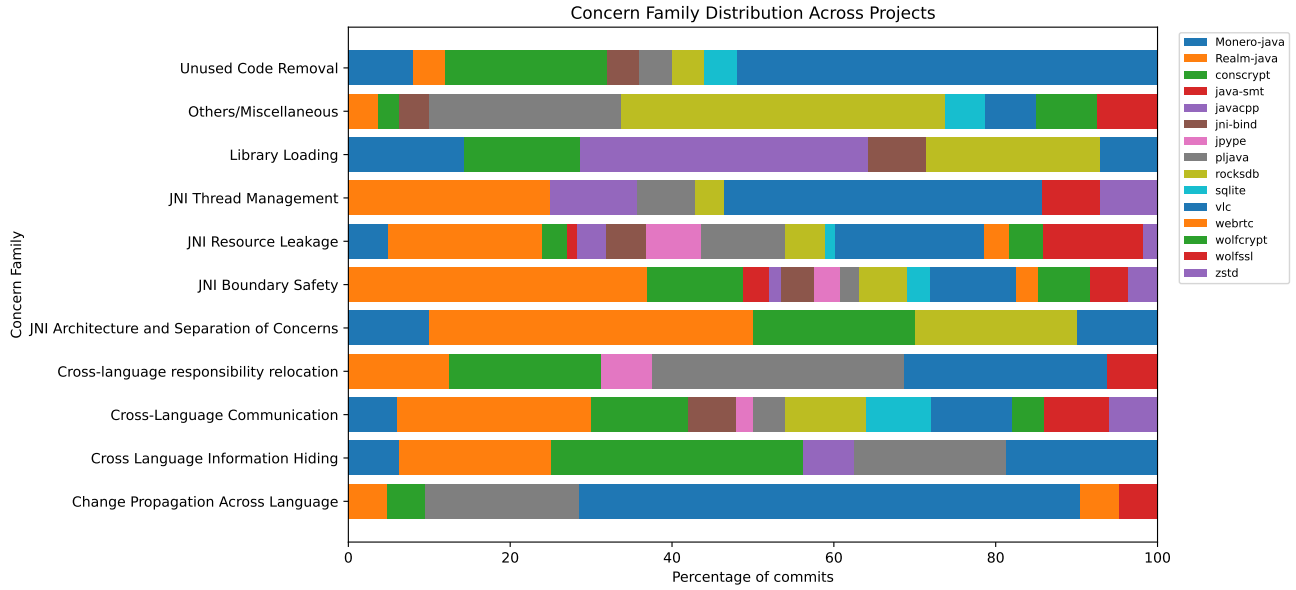


Figure 2: Project-wise percentage distribution of the extracted JNI maintenance concern families. Each horizontal stacked bar represents one concern family, showing the relative contribution of each analyzed project to the commits belonging to that concern family.

- Library Loading: 14 commits (2.18%), and
- JNI Architecture and Separation of Concerns: 10 commits (1.56%).

Although individually less frequent, these concern families collectively demonstrate that JNI maintenance extends beyond low-level correctness issues to include architectural restructuring, interface evolution, modularization, library management, and cross-language design improvements.

Distribution Across Projects

While Table 2 summarizes the overall frequency of the extracted concerns, frequency alone is insufficient to establish that a concern represents a general JNI maintenance problem. A concern observed only within one or two repositories may simply reflect project-specific implementation choices, architectural characteristics, or maintenance practices rather than a recurring challenge encountered across JNI-based software. Therefore, in addition to reporting the overall distribution, it is important to examine how the identified concern families are distributed across independent projects.

Figure 2 presents the project-wise distribution of the extracted concerns.

The stacked bar chart shows that the majority of the extracted concern families are distributed across multiple independent repositories rather than being concentrated within a single project. In particular, the two dominant concern families, *JNI Boundary Safety* and *JNI Resource Leakage*, are observed throughout the analyzed project corpus and are especially prominent in large JNI-intensive systems such

as Realm-java, VLC, and Wolfssl. Their widespread occurrence suggests that these concerns represent common maintenance challenges inherent to JNI development rather than project-specific issues.

Similarly, although architectural concern families such as *Cross-language Responsibility Relocation*, *Cross-Language Information Hiding*, and *JNI Architecture and Separation of Concerns* occur less frequently, they nevertheless appear across multiple repositories. Their recurrence in independent projects provides evidence that these concerns arise repeatedly in practice despite their comparatively lower frequency.

The figure also reveals differences in maintenance emphasis among projects. Some repositories devote a larger proportion of their JNI maintenance effort to boundary safety and resource management, whereas others exhibit comparatively greater attention to communication, architectural restructuring, responsibility relocation, or library management. These differences likely reflect variations in application domains, software architecture, JNI usage patterns, and project maturity.

Overall, the project-wise distribution strengthens the validity of the extracted concern taxonomy by demonstrating that the identified concern families recur across multiple independent JNI systems rather than being artifacts of individual repositories. This cross-project recurrence provides greater confidence that the extracted concerns represent general JNI maintenance challenges and therefore constitute an appropriate empirical basis for the subsequent smell-mapping analysis.

Overall, the findings answer RQ1 by showing that developers repeatedly address a diverse set of JNI-related

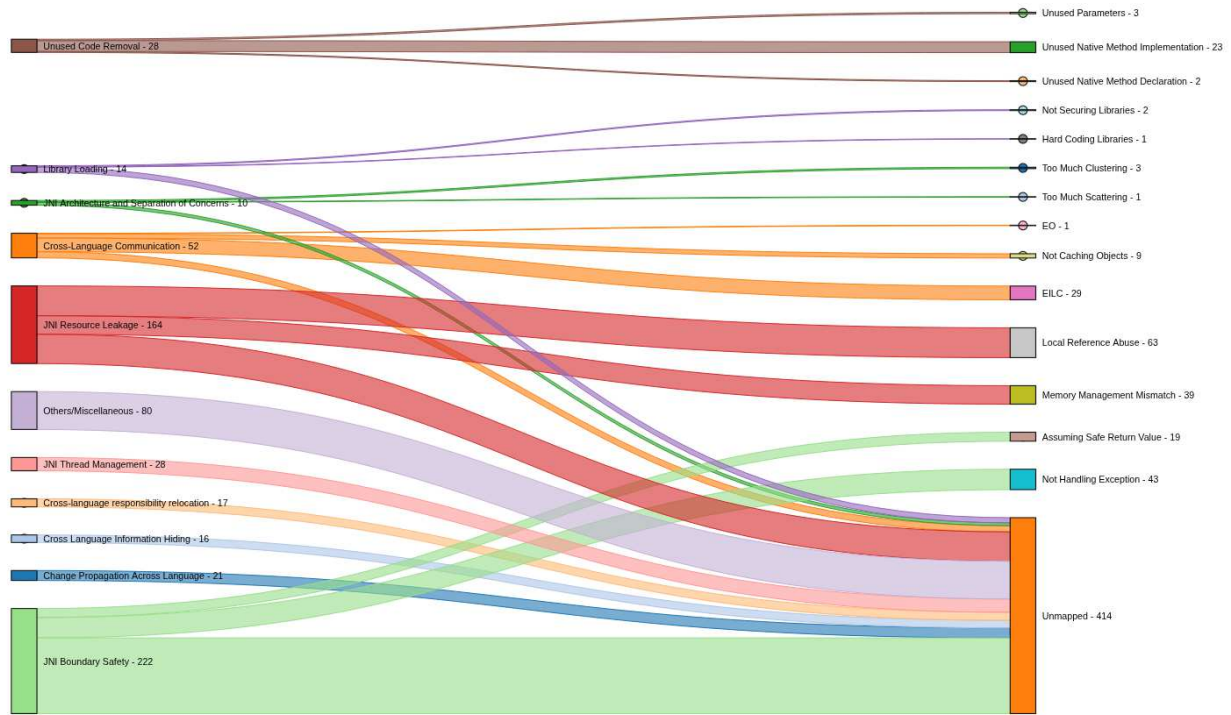


Figure 3: Mapping between extracted JNI maintenance concern families and existing JNI code and design smells. The width of each flow represents the number of manually verified commits supporting the corresponding mapping.

maintenance concerns during software evolution. While correctness and resource-management issues dominate maintenance activities, the observed concern families span multiple aspects of JNI development, including communication, architecture, responsibility allocation, interface evolution, threading, and library management. These extracted concern families provide the empirical foundation for the subsequent analysis, where we investigate their relationship with existing JNI code and design smells (RQ2).

4.2. RQ2: Which extracted maintenance concerns are adequately represented by existing JNI code and design smells?

To answer RQ2, following the procedure described in Section 3.4, each maintenance concern identified in RQ1 was systematically mapped to the existing JNI code and design smell catalog whenever the underlying maintenance activity corresponded to a previously defined smell. Maintenance activities for which no existing smell adequately captured the addressed concern were classified as *Unmapped*. Figure 3 illustrates the relationship between the extracted maintenance concern families and the corresponding JNI smell categories.

The mapping reveals that the existing JNI smell catalog provides only partial coverage of the maintenance concerns addressed by developers. Among the eleven identified concern families, only two concern families (*JNI*

Resource Leakage and *Unused Code Removal*) are well represented by existing JNI smells, with the majority of their maintenance activities mapping to previously defined smell categories. Four concern families (*JNI Boundary Safety*, *Cross-Language Communication*, *Library Loading*, and *JNI Architecture and Separation of Concerns*) are only partially represented, where some maintenance activities correspond to existing smells while a substantial proportion remain unmapped. The remaining five concern families (*Change Propagation Across Language*, *Cross-Language Information Hiding*, *Cross-language Responsibility Relocation*, *JNI Thread Management*, and *Others/Miscellaneous*) receive no meaningful representation from the current JNI smell catalog, as their maintenance activities could not be consistently associated with any existing smell definition.

Overall, of the 652 concern-addressing commits, 238 commits (36.50%) could be mapped to one or more existing JNI smells, whereas the remaining 414 commits (63.50%) remained unmapped. These findings indicate that although existing JNI smells capture several recurring implementation-level maintenance concerns, they explain only a limited proportion of the maintenance activities observed during software evolution.

Among the mapped smells, *Local Reference Abuse* receives the strongest empirical support, accounting for 63 concern-addressing commits. All of these commits originate

from the *JNI Resource Leakage* concern and involve improvements to JNI reference management, resource cleanup, and native object lifetime management. This finding indicates that improper management of JNI references and resources remains one of the most frequently addressed maintainability issues represented by the current JNI smell catalog.

The second most strongly supported smell is *Not Handling Exception*, which is supported by 43 commits from the *JNI Boundary Safety* concern family. These maintenance activities primarily introduce explicit exception checking, exception propagation, and failure handling across the Java-native boundary, highlighting the importance of robust error handling in JNI applications.

Similarly, *Memory Management Mismatch* receives support from 39 commits, again originating from the *JNI Resource Leakage* concern family. These commits correct inconsistencies in resource allocation and deallocation, demonstrating recurring maintenance effort devoted to ensuring proper ownership and lifetime management of native resources.

Communication-related maintenance activities provide strong evidence for *Excessive Inter-Language Communication (EILC)*, which is supported by 29 commits. These commits originate exclusively from the *Cross-Language Communication* concern and primarily reduce unnecessary interactions across the Java-native boundary, thereby improving maintainability and reducing cross-language dependencies.

The *Unused Native Method Implementation* smell is supported by 23 commits from the *Unused Code Removal* concern family, reflecting recurring maintenance activities that remove obsolete or redundant native implementations no longer required by the Java layer.

Assuming Safe Return Value is supported by 19 commits, all originating from the *JNI Boundary Safety* concern family. These commits improve the robustness of JNI interactions by introducing appropriate validation of returned values before subsequent processing.

Several smells receive comparatively limited empirical support. *Not Caching Objects* is supported by 9 commits from the *Cross-Language Communication* concern family, while *Too Much Clustering* and *Unused Parameters* are each supported by three commits originating from *JNI Architecture and Separation of Concerns* and *Unused Code Removal*, respectively. Likewise, *Not Securing Libraries* receives support from two commits associated with the *Library Loading* concern family, whereas *Unused Native Method Declaration* is supported by two commits involving the removal of obsolete native declarations. Finally, *Hard Coding Libraries*, *Too Much Scattering*, and *EO* each receive evidence from a single maintenance activity.

It is important to note that the amount of maintenance evidence supporting a smell should not be interpreted as a measure of its prevalence or importance. Our study analyzes maintenance activities rather than the occurrence of smell instances. Consequently, a smell supported by relatively few maintenance commits may still be prevalent in practice

but remain unaddressed, be difficult or costly to refactor, or simply receive less maintenance attention. Conversely, smells with stronger empirical support indicate concerns that developers repeatedly invest effort in addressing during software evolution. Therefore, the reported commit counts should be interpreted as evidence of maintenance activity rather than as a ranking of smell significance.

Figure 3 illustrates how the extracted maintenance concern families are represented by the existing JNI smell catalog. It highlights three distinct situations: concern families that are well represented by existing smells, concern families that are only partially represented, and concern families that remain entirely uncovered. These observations provide a clear picture of the empirical coverage of the current JNI smell catalog and identify the areas requiring further investigation.

Overall, the mapping results indicate that the current JNI smell catalog captures several recurring implementation-level maintenance concerns, particularly those related to resource management, exception handling, and cross-language communication. However, a substantial proportion of the identified maintenance activities remain either only partially represented or completely uncovered by the existing smell definitions. These unmapped concern families are examined further in RQ3 to investigate whether they represent previously uncharacterized JNI-specific code and design smells.

4.3. RQ3: Which recurring maintenance concerns that are not adequately represented by existing JNI smell catalogs motivate new JNI-specific smell definitions?

The results of RQ2 revealed that a substantial proportion of the extracted maintenance concerns could not be adequately represented by existing JNI code and design smells. However, the absence of an existing smell does not automatically justify proposing a new one. A code or design smell should represent a recurring and conceptually coherent maintainability problem rather than isolated or unrelated maintenance activities. Consequently, only unmapped concern families exhibiting recurring evidence of the same underlying design deficiency were considered as candidates for new smell definitions.

Following the procedure mentioned in Section 3.5, our analysis identified two categories of unmapped concerns. The first category consists of partially represented concern families, where existing smell definitions explain only a subset of the observed maintenance activities. For example, JNI Boundary Safety and JNI Resource Leakage are partially represented by smells such as Not Handling Exception, Assuming Safe Return Value, Local Reference Abuse, and Memory Management Mismatch. Although these concern families contain additional unmapped maintenance activities, the remaining instances correspond to diverse implementation issues rather than a single recurring design deficiency. Therefore, they do not provide sufficient evidence for proposing additional smells.

The second category consists of largely or completely unmapped concern families. These concerns were further analyzed to determine whether they exhibit recurring maintenance patterns that consistently indicate a previously uncharacterized cross-language design deficiency.

Among these concerns, JNI Thread Management represents an important maintenance activity but does not exhibit sufficient conceptual consistency. The analyzed commits addressed a diverse collection of issues, including thread attachment, thread detachment, synchronization, execution contexts, and thread-local storage. Although these issues recur, they represent different maintenance problems rather than manifestations of a single underlying design deficiency. Consequently, JNI Thread Management does not currently justify the definition of a new smell.

In contrast, three concern families consistently exhibited recurring patterns corresponding to well-defined cross-language design deficiencies.

The first concern, *Cross-Language Change Propagation*, repeatedly revealed coordinated modifications spanning Java and native code. These changes commonly involved updating method signatures, class names, JNI bindings, and interface definitions to preserve consistency across the language boundary. The repeated occurrence of such synchronized modifications indicates that explicit cross-language dependencies reduce change locality and increase maintenance effort.

The second concern, *Cross-Language Information Hiding*, repeatedly involved maintenance activities aimed at reducing the exposure of implementation details across language boundaries. Developers introduced abstraction layers, encapsulated native functionality, and reduced direct exposure of implementation-specific interfaces, indicating recurring violations of abstraction boundaries between Java and native code.

The third concern, *Cross-Language Responsibility Relocation*, repeatedly involved relocating responsibilities between Java and native code to improve modularity, reduce coupling, and place behavior closer to the language where it naturally belongs. These maintenance activities indicate recurring instances of misplaced behavioral responsibilities across the language boundary.

To summarize the empirical evidence supporting the proposed smell definitions, Table 3 relates each candidate smell to its motivating maintenance concern, the extent of its recurrence, the violated software design principle, and the corresponding coverage by the existing JNI smell catalog. Together, these observations demonstrate that the proposed smells are not isolated maintenance activities but recurring cross-language design deficiencies that are consistently observed across multiple repositories and are not adequately represented by existing JNI smell definitions.

Table 3 shows that all three proposed smells satisfy the candidate-smell identification criteria defined in Section 3.5. Each concern recurs across multiple commits and independent projects, consistently reflects a single underlying cross-language design deficiency, violates a well-established

software design principle, and cannot be adequately represented by the existing JNI smell catalog. Consequently, these concerns provide sufficient empirical and conceptual evidence to motivate the proposal of three candidate JNI-specific smell definitions:

1. Cross-Language Abstraction Leakage (CLAL), motivated by recurring violations of *Cross-Language Information Hiding*.
2. Wrong Responsibility Allocation (WRA), motivated by recurring *Cross-Language Responsibility Relocation*.
3. Cross-Language Shotgun Surgery (CLSS), motivated by recurring *Cross-Language Change Propagation*.

The complete conceptual definitions, representative manifestations, associated problems, initial operationalizations, and refactoring recommendations for these smells are presented in the next section.

5. Smell Definitions

This section defines the candidate multi-language code smells in detail. Each smell is described in terms of its motivation, the involved design concern, the cross-language manifestation of the design concern, problem, the manifestation in JNI setup and example, detection rationale, initial operationalization of the detection rule, and recommended refactoring.

5.1. Cross-Language Abstraction Leakage (CLAL)

Empirical Motivation: The concern *Cross-Language Information Hiding* repeatedly contained maintenance activities that strengthened abstraction boundaries across Java and native code. Developers consistently introduced wrappers, reduced exposure of implementation details, and encapsulated JNI-specific functionality behind well-defined interfaces. These recurring maintenance activities indicate that abstraction boundaries are frequently weakened across language boundaries.

Design Concern: Abstraction and information hiding are fundamental software design principles that promote modularity by concealing implementation details behind well-defined interfaces. By exposing only the functionality required by clients while hiding internal representations, software components can evolve independently with minimal impact on other parts of the system, thereby reducing coupling and improving maintainability Parnas (1972); Meyer (1997).

Cross-Language Manifestation: In multi-language systems, interoperability mechanisms introduce communication paths across language boundaries. If these mechanisms expose implementation details rather than stable abstractions, components implemented in one language become dependent on the internal structure of components implemented in another. Consequently, abstraction boundaries weaken, increasing cross-language coupling and reducing modularity.

Table 3

Empirical evidence supporting the proposed JNI-specific smell definitions.

Candidate Smell	Motivating Concern	Commits	Projects	Underlying Design Principle	Existing Smell Coverage
Cross-Language Shotgun Surgery (CLSS)	Cross-Language Change Propagation	21	6	Change locality	None
Cross-Language Abstraction Leakage (CLAL)	Cross-Language Information Hiding	16	6	Abstraction and Information Hiding	None
Wrong Responsibility Allocation (WRA)	Cross-Language Responsibility Relocation	17	6	GRASP Information Expert (Responsibility Assignment)	None

Problem: *Cross-Language Abstraction Leakage (CLAL)* occurs when implementation details that should remain hidden within one language become unnecessarily exposed across the language boundary. Such exposure weakens abstraction boundaries by allowing software entities implemented in one language to depend directly on internal state, behavior, or implementation-specific details of another language.

The presence of *CLAL* may lead to:

- Violation of abstraction and information hiding across language boundaries.
- Increased cross-language coupling due to dependencies on implementation details.
- Reduced maintainability, as modifications to internal structures propagate across languages.
- Increased risk of inconsistencies, security vulnerabilities, or privacy violations when internal state becomes externally accessible.

JNI Manifestation and Example: One representative manifestation of *CLAL* in JNI-based systems occurs when native code directly accesses Java members declared as *private* or *protected*. Although these members are intentionally hidden within the Java language, JNI functions such as *GetFieldID()*, *GetMethodID()*, and *GetObjectField()* enable native code to access them directly, thereby exposing implementation details across the language boundary.

Figures 1 and 2 illustrate this situation. The Java class declares the field *password* as private, yet the native implementation retrieves and accesses it through JNI. Consequently, the abstraction boundary established by the Java language is weakened by the interoperability layer.

```
package com.example
public class Printer {
    public static native void printWithClass();
    private String password;
}
}
```

Listing 1: Example of smell CLAL

```
JNIEXPORT void JNICALL Java_com_example_Printer_printWithClass
(
    JNIEnv* env, jclass clazz) {
    // get class reference
    jclass cls = (*env)->FindClass(env, "com/example/Printer");
    // get the ID of the private field
    jfieldID fid = env->GetFieldID(cls, "password", "Ljava/lang/
        String;");
    // get the value of the field
    jobject val = env->GetObjectField(obj, fid);
}
```

Listing 2: Example of smell CLAL

Detection Rationale: The proposed detection rule operationalizes *CLAL* using one representative manifestation of the smell. Specifically, it identifies situations where native code directly accesses non-public Java fields or methods. Such accesses indicate that implementation details intended to remain hidden are exposed across the language boundary, creating explicit dependencies on internal program structures.

This manifestation was selected because it is both observable and common in JNI applications, making it suitable for automated detection and empirical validation. However, it does not exhaustively characterize *CLAL*. Other forms of abstraction leakage—including exposing implementation-specific native interfaces, bypassing abstraction layers, leaking internal object representations, or relying on implementation-dependent JNI interfaces—may also constitute manifestations of the same underlying design deficiency and can be incorporated into future detection techniques.

Initial Operationalization:

$$\begin{aligned}
 &\exists C \in \text{JavaClasses} : \\
 &\quad \left(\exists m \in \text{Members}(C) : \text{visibility}(m) \neq \text{public} \right) \\
 &\quad \wedge \\
 &\quad \left(\exists f \in \text{NativeCalls} : f \in \{\text{FindClass}, \text{GetObjectClass}\} \right. \\
 &\quad \quad \left. \wedge \text{resolvesTo}(\text{argAt}(f, \text{classPos}), C) \right)
 \end{aligned}$$

In other words, if a native call resolves to a Java class whose non-public members are accessed, the file is flagged

as exhibiting EB.

Refactoring: To mitigate *CLAL*, interactions across the language boundary should occur through well-defined public interfaces rather than implementation-specific details. Direct access to non-public fields and methods should be avoided in favor of stable APIs, wrapper classes, or abstraction layers that preserve information hiding. Strengthening abstraction boundaries reduces cross-language coupling, improves modularity, and enables both language components to evolve independently.

5.2. Wrong Responsibility Allocation (WRA)

Empirical Motivation: Our empirical analysis repeatedly identified commits that relocated functionality between Java and native code to improve modularity, reduce unnecessary language transitions, and place responsibilities closer to their primary collaborators. These recurring maintenance activities indicate that behavioral responsibilities are sometimes implemented in the language that is not their natural location.

Design Concern: According to the General Responsibility Assignment Software Patterns (GRASP) Larman (2012), responsibilities should be assigned to the software component that possesses the information and behavior required to fulfill them, thereby promoting high cohesion and low coupling. When responsibilities are allocated close to their primary collaborators and supporting functionality, software components become easier to understand, maintain, and evolve. Conversely, assigning responsibilities to components that primarily depend on externally implemented functionality reduces behavioral cohesion, increases coupling, and obscures the natural decomposition of the system.

Cross-Language Manifestation: In multi-language systems, responsibilities may be distributed across different implementation languages. Ideally, a responsibility should reside in the language where the majority of its collaborators and behavioral dependencies exist. However, language boundaries may separate responsibilities from the functionality upon which they primarily depend, resulting in misplaced behavior and unnecessary cross-language interactions.

Problem: *Wrong Responsibility Allocation (WRA)* occurs when a software responsibility is implemented in one language even though most of the functionality required to realize that responsibility resides in another language. Consequently, the implementation primarily orchestrates foreign-language behavior rather than contributing meaningful behavior within its own language, resulting in reduced behavioral cohesion and increased cross-language coupling.

The presence of *WRA* may lead to:

- Reduced behavioral cohesion due to misplaced responsibilities.
- Increased semantic coupling between language layers.
- Reduced maintainability, as developers must reason about behavior distributed across multiple languages.

- Additional runtime overhead caused by excessive inter-language invocations Kurzyniec and Sunderam (2001).
- Architectural inconsistency arising from responsibilities being implemented away from their natural collaborators.

JNI Manifestation and Example: One representative manifestation of *WRA* in JNI systems occurs when a non-native Java method primarily invokes native methods instead of utilizing functionality implemented within the Java layer (or conversely when native code primarily orchestrates Java functionality). Although the method is implemented in one language, most of the behavior required to perform its responsibility resides in the other language.

Figure 3 illustrates this situation. The method *process()* contributes little Java-specific behavior and primarily delegates work to native methods. Consequently, the responsibility implemented by *process()* is more naturally associated with the native layer than with the Java layer in which it is declared.

```
public class Encryptor {
    public native String encrypt(String text);
    public native String decrypt(String text);

    // BAD: Relies almost entirely on native logic
    public String process(String data) {
        String enc = encrypt(data);
        return decrypt(enc);
    }
}
```

Listing 3: Example of the smell *WRA*

Detection Rationale: The proposed detection rule operationalizes *WRA* using one representative manifestation of the smell. Specifically, it identifies non-native Java methods whose behavior is dominated by invocations of native methods. Such methods primarily orchestrate functionality implemented in another language rather than contributing substantial local behavior, suggesting that the responsibility may be allocated to the wrong side of the language boundary.

This manifestation was selected because it is both observable and common in JNI systems, making it suitable for automated detection and empirical validation. However, it does not exhaustively characterize *WRA*. Other manifestations—including misplaced native implementations, responsibilities driven predominantly by cross-language data dependencies, or other forms of behavioral misallocation across language boundaries—may also constitute instances of the same underlying design deficiency and can be incorporated into future detection techniques.

Initial Operationalization: The proposed operationalization assesses responsibility allocation by comparing a method's interactions with locally implemented functionality against those implemented in the foreign language. This intuition is captured using the *Locality of Method Accesses (LMA)* metric, defined as the ratio of local method accesses to the total

number of method accesses. A low LMA indicates that the method primarily relies on foreign-language functionality, suggesting that the responsibility may be more appropriately implemented on the other side of the language boundary.

Following the Feature Envy detection strategy implemented in iPlasma Cristina, Radu, Mihancea et al. (2005), a method is considered a candidate instance of WRA if it performs more than two foreign-language method accesses ($ATMFL > 2$) while fewer than one-third of its total method accesses are local ($LMA < \frac{1}{3}$):

$$IsWRA(m) = ATMFL(m) > FEW_{ATMFL} \wedge LMA(m) < MAX_PERCENT_{LMA}$$

where $ATMFL$ (*Access To Methods in Foreign Language*) denotes the number of accesses to methods implemented in a different programming language, and

$$LMA = \frac{\#Local\ Method\ Accesses}{\#Total\ Method\ Accesses\ (Local+foreign)}$$

Following iPlasma Cristina et al. (2005), the thresholds are set to $FEW_{ATMFL} = 2$ and $MAX_PERCENT_{LMA} = \frac{1}{3}$.

Refactoring: To mitigate WRA, responsibilities should be relocated closer to the language where their primary collaborators and behavioral dependencies reside. If a method primarily orchestrates foreign-language functionality, the responsibility should be migrated to the foreign component and exposed through an appropriate interface. Such relocation improves behavioral cohesion, reduces unnecessary language transitions, decreases semantic coupling, and aligns implementation with architectural responsibilities.

5.3. Cross-Language Shotgun Surgery (CLSS)

Empirical Motivation: Our empirical analysis repeatedly identified commits in which modifications to Java artifacts required corresponding changes in native code, and vice versa. These coordinated modifications occurred across multiple repositories and typically involved hardcoded class names, method signatures, JNI bindings, and interface definitions. Such recurring change propagation indicates that explicit cross-language dependencies reduce change locality and increase maintenance effort.

Design Concern: Software maintainability depends on *change locality*, i.e., the ability to confine modifications to a small and localized portion of the system. When a seemingly simple change requires coordinated updates across multiple artifacts, maintenance effort increases, the likelihood of inconsistencies grows, and future evolution becomes more difficult. This principle underlies the well-known *Shotgun Surgery* smell, in which a single logical change propagates across multiple implementation locations Fowler, Beck, Brant, Opdyke and Roberts (2002). Preserving change locality by minimizing unnecessary change propagation is therefore a fundamental design objective for achieving maintainable and evolvable software Winter (2014).

Cross-Language Manifestation: In multi-language systems, interoperability mechanisms introduce explicit dependencies between software artifacts implemented in different languages. Consequently, modifications that would otherwise remain localized may propagate across language boundaries, requiring synchronized changes in multiple implementation environments. The language boundary therefore becomes an additional source of change propagation and maintenance complexity.

Problem: *Cross-Language Shotgun Surgery (CLSS)* occurs when explicit cross-language dependencies repeatedly force modifications in one language to be accompanied by corresponding modifications in another language. As a result, changes that should ideally remain localized become distributed across language boundaries, increasing maintenance effort, reducing change locality, and making software evolution more error-prone.

The presence of CLSS may lead to:

- Reduced change locality due to coordinated modifications across multiple languages.
- Increased maintenance effort resulting from synchronized updates to heterogeneous artifacts.
- Increased risk of inconsistencies and runtime failures when dependent artifacts evolve independently.
- Reduced evolvability, as developers may avoid beneficial changes because of the associated cross-language maintenance cost.

JNI Manifestation and Example: In JNI-based systems, one common manifestation of CLSS occurs when Java class names or method signatures are hardcoded within native code. Since JNI resolves Java artifacts through string-based identifiers, changes to Java packages, class names, or method signatures require corresponding modifications in the native implementation.

Figure 4 illustrates this situation. Both the fully qualified Java class name and the corresponding method signature are embedded as string literals in native code. Consequently, changes to either artifact require synchronized updates across both Java and native code, reducing change locality.

```
JNIEXPORT void JNICALL Java_com_example_Printer_printWithClass
(
    JNIEnv* env, jclass clazz) {
    // BAD: Hardcoded class reference
    jclass cls = (*env)->FindClass(env, "com/example/Printer");

    // BAD: Hardcoded method signature
    jmethodID method = (*env)->GetMethodID(env, cls, "method1", "(
        Ljava/lang/String;)V");
}
```

Listing 4: Example of the smell CLSS

Detection Rationale: The proposed detection rule operationalizes CLSS using one representative manifestation of

the smell. Specifically, it identifies hardcoded Java class names and method signatures embedded within native code, as these introduce explicit cross-language dependencies that directly couple artifacts across the language boundary.

This manifestation was selected because it is both observable and frequently encountered in JNI systems, making it suitable for automated detection and empirical validation. However, it is not intended to exhaustively characterize CLSS. Other forms of explicit cross-language dependencies—including hardcoded field identifiers, native registration metadata, generated bindings, framework-specific interface definitions, or other mechanisms that require synchronized cross-language modifications—may also constitute manifestations of the same underlying design deficiency and can be incorporated into future detection techniques.

Initial Operationalization.

$$\begin{aligned} \exists f \in \text{NativeCalls} : \\ & (f = \text{FindClass} \wedge \text{isStringLiteral}(\text{argAt}(f, \text{classPos}))) \\ \vee \\ & (f \in \{\text{GetMethodID}, \text{GetStaticMethodID}\} \\ & \wedge \text{isStringLiteral}(\text{argAt}(f, \text{signaturePos}))) \end{aligned}$$

That is, if the argument passed at the `classPos` position of `FindClass`, or the `signaturePos` position of `GetMethodID` or `GetStaticMethodID`, is a string literal, the file is marked as exhibiting CLSS.

Refactoring: To mitigate CLSS, cross-language interfaces should minimize explicit dependencies on implementation-specific identifiers. Rather than embedding class names and method signatures as string literals, systems should employ abstraction mechanisms such as generated bindings, centralized interface definitions, reflection-based approaches, or APIs that provide class and method references directly. Reducing explicit cross-language dependencies improves change locality by minimizing the number of artifacts that must be modified in response to a single change.

6. Discussion

The findings of this study provide new insights into how developers maintain JNI-based systems and how these maintenance activities relate to existing JNI code and design smells. Rather than identifying smells directly from source code, our study investigates the maintenance concerns that developers repeatedly address during software evolution. This perspective enables us to assess whether the existing JNI smell catalog adequately represents the practical problems encountered by developers.

6.1. Existing JNI Smell Catalogs Capture Only Part of Developer Maintenance Concerns

One of the principal findings of this study is that software evolution history provides an effective means of empirically evaluating existing multi-language smell catalogs. Using JNI as the target interoperability technology, we found that

the current JNI smell catalog provides only partial coverage of the maintenance concerns repeatedly addressed by developers. While six of the eleven identified concern families are represented either fully or partially, only 35.4% of the manually verified concern-addressing commits could be mapped to existing JNI code and design smells. The remaining maintenance activities correspond to recurring concerns that are either only partially characterized or completely absent from the current smell catalog.

Interestingly, the smells receiving the strongest empirical support predominantly capture implementation-level concerns, including resource management, exception handling, and excessive cross-language communication. These issues are directly observable in source code and have consequently received considerable attention in previous smell catalogs. In contrast, several recurring architectural concerns, such as cross-language change propagation, abstraction leakage, and responsibility allocation, remain largely unrepresented despite repeatedly motivating maintenance activities across multiple repositories.

The observed difference between well-represented and uncovered concern families is particularly noteworthy. Concern families that are well represented by the existing JNI smell catalog primarily correspond to implementation-level issues, such as resource management, exception handling, and unnecessary cross-language communication. These deficiencies are localized, directly observable in source code, and therefore naturally lend themselves to static analysis and rule-based smell detection. In contrast, the uncovered concern families are predominantly architectural in nature, involving change propagation, abstraction boundaries, and responsibility allocation across the Java-native interface. Such concerns arise from the interaction between software components during evolution rather than from isolated implementation constructs. This distinction provides a plausible explanation for why the current JNI smell catalog captures implementation-level deficiencies more effectively than recurring architectural maintenance concerns.

These findings should not be interpreted as shortcomings of previous JNI smell research. Rather, they reflect the complementary perspectives adopted by existing studies and the present work. Existing JNI smell catalogs were primarily derived from software design principles and expert knowledge, whereas our study evaluates those definitions using empirical evidence from software evolution. Consequently, software evolution reveals recurring maintainability concerns that complement design-based reasoning and exposes gaps that are difficult to identify through static analysis alone.

6.2. Maintenance Concerns as an Empirical Basis for Smell Discovery

Traditional code and design smell research predominantly follows a top-down process in which candidate smells are derived from software design principles and subsequently operationalized through detection rules. Our

study complements this paradigm by introducing a bottom-up, concern-centric methodology in which recurring maintenance concerns are first mined from software evolution history and then systematically related to existing smell definitions.

This concern-centric perspective provides two advantages. First, it enables an empirical assessment of the practical relevance of existing smell catalogs by evaluating whether developers repeatedly invest maintenance effort in addressing the underlying concerns. Second, it provides a systematic mechanism for identifying gaps in existing smell catalogs. Rather than proposing new smells solely from theoretical reasoning, candidate smells emerge from recurring maintenance activities that exhibit conceptual consistency, recur across independent projects, and cannot be adequately explained by existing smell definitions.

More broadly, these findings suggest that software evolution history represents a valuable complementary source of evidence for code and design smell research. While design principles remain essential for defining maintainability problems, maintenance history provides empirical evidence regarding which problems repeatedly affect developers in practice.

Although this study focuses on JNI, the proposed concern-centric methodology is applicable to any multi-language ecosystem in which software evolution history can be mined and maintenance concerns can be related to existing smell definitions.

6.3. Implications of the Proposed JNI Smells

The three proposed candidate smells, Cross-Language Shotgun Surgery (CLSS), Cross-Language Abstraction Leakage (CLAL), and Wrong Responsibility Allocation (WRA), illustrate how software evolution history can reveal recurring cross-language design deficiencies that are not adequately represented by existing smell catalogs. Unlike many existing JNI smells, which primarily capture implementation-level interoperability issues, the proposed smells focus on architectural properties governing change locality, abstraction boundaries, and responsibility allocation across language interfaces.

An important characteristic of the proposed smells is that the presented operationalizations should not be interpreted as exhaustive detection rules. Instead, they represent one observable manifestation of a broader underlying design deficiency derived from recurring maintenance evidence. Additional manifestations may exist and should be incorporated as further empirical evidence becomes available. Consequently, the proposed smells should be regarded as candidate JNI-specific smell definitions whose operationalizations can evolve alongside future empirical studies.

6.4. Implications

For researchers, this study demonstrates that software evolution history provides a valuable complementary source of evidence for evaluating and extending existing multi-language code and design smell catalogs. Although demonstrated using JNI systems, the proposed concern-centric

methodology is independent of the underlying interoperability technology and may be applied to other multi-language ecosystems, including Java Native Access (JNA), Python C extensions, JavaScript–WebAssembly systems, and mixed-language mobile applications. The proposed methodology therefore offers a general framework for empirically validating existing smell definitions and identifying new smell candidates from software evolution.

For practitioners, the extracted maintenance concerns highlight recurring problems that repeatedly consume developer effort during the evolution of JNI applications. Awareness of these concerns can assist developers in designing more maintainable cross-language interfaces and identifying architectural weaknesses before they manifest as recurring maintenance activities. Integrating the proposed smells into static analysis tools and continuous integration pipelines may enable earlier identification of cross-language maintainability problems, thereby reducing long-term maintenance costs.

Overall, this study demonstrates that software evolution history complements traditional design-based approaches to smell discovery. While software design principles remain fundamental for identifying maintainability problems, recurring maintenance concerns provide empirical evidence of the problems that developers repeatedly encounter in practice. Combining these complementary perspectives offers a more comprehensive foundation for evaluating and extending multi-language code and design smell catalogs.

7. Threats to Validity

Like any empirical software engineering study, our work is subject to several threats to validity.

7.1. Construct Validity

Construct validity concerns whether the employed methodology accurately captures the concepts under investigation. Our study relies on an LLM-assisted workflow to identify JNI-related maintenance concerns from commit messages and commit diffs. Although recent studies have demonstrated the effectiveness of Large Language Models for source-code understanding and commit-level reasoning, incorrect interpretations remain possible. To assess the reliability of the concern extraction process, we randomly selected 10% of the identified concern-addressing commits for manual validation. Among the 65 manually inspected commits, 63 were confirmed to be correctly classified, yielding a precision of 96.92%. Since the objective of this study is to obtain reliable evidence of maintenance concerns rather than exhaustively identify every concern-addressing commit, we emphasize precision rather than recall. Consequently, some concern-addressing commits may remain unidentified, but this does not affect the validity of the extracted concern taxonomy.

Another potential threat arises from the manual mapping of maintenance concerns to existing JNI smell definitions. Because the mapping involves conceptual interpretation, it

is inherently subjective. To mitigate this threat, two authors independently mapped each maintenance concern to the existing JNI smell catalog. The mapping achieved a Cohen’s κ coefficient of 0.913, indicating almost perfect agreement Landis and Koch (1977). Remaining disagreements were conservatively classified as *Unmapped*, thereby reducing the likelihood of overstating the empirical coverage of the existing JNI smell catalog.

Finally, the proposed JNI smell definitions are operationalized using representative manifestations observed in the analyzed repositories. These operationalizations capture one empirically supported manifestation of each underlying design deficiency and should not be interpreted as exhaustive detection rules. Additional manifestations may be identified through future empirical studies.

7.2. Internal Validity

Internal validity concerns factors that may influence the observed results. The identification of maintenance concerns depends on the information available in commit messages and commit diffs. Developers may omit important contextual information from commit messages, and individual commits may contain multiple maintenance activities, making interpretation more challenging.

Furthermore, the study intentionally focuses on commits providing evidence of recurring JNI-related maintenance concerns. Feature additions, architectural extensions, documentation updates, build modifications, and other development activities that did not explicitly address existing maintenance concerns were excluded. Although this conservative filtering improves the reliability of the extracted evidence, some relevant maintenance activities may have been excluded because of ambiguous commit histories or insufficient contextual information.

7.3. External Validity

External validity concerns the generalizability of the findings. The study analyzes the complete development histories of fifteen open-source JNI projects spanning multiple application domains. Although these projects differ in size, maturity, and development characteristics, the extracted concern taxonomy and the proposed smell definitions may not generalize directly to proprietary software systems or to software employing alternative interoperability technologies such as Java Native Access (JNA), SWIG, or WebAssembly-based frameworks.

Nevertheless, the proposed concern-centric methodology is not inherently limited to JNI. The methodology of mining maintenance concerns from software evolution and evaluating their correspondence to existing smell catalogs can potentially be applied to other multi-language ecosystems. However, whether similar maintenance concerns and smell definitions arise in those environments remains an open question requiring further empirical investigation.

7.4. Reliability

Reliability concerns the reproducibility of the study. To improve repeatability, the entire analysis followed a

predefined workflow consisting of project selection, JNI-related commit extraction, LLM-assisted concern identification, manual validation, concern taxonomy construction, independent concern-to-smell mapping, and identification of candidate JNI smell definitions. The same prompting strategy was used throughout the LLM-assisted analysis, with only the commit message and commit diff varying between executions, ensuring consistent treatment of all analyzed commits.

We provide all the data, results, and the prompt used in this study in the online replication package ².

8. Conclusion

Multi-language software systems introduce unique maintenance challenges that are not adequately captured by traditional single-language code smells. Although previous research has proposed catalogs of multi-language code and design smells, particularly for JNI-based systems, these catalogs have largely been derived from software design principles and evaluated through static analysis. Consequently, it has remained unclear whether the proposed smell definitions correspond to the maintenance concerns that developers repeatedly encounter and address during software evolution.

This paper presented a concern-centric methodology for empirically evaluating multi-language smell catalogs through software evolution history, using JNI as a representative multi-language interoperability technology. By combining repository mining, LLM-assisted commit-diff analysis, manual validation, and independent smell mapping, we analyzed the maintenance history of fifteen open-source JNI projects. The study identified eleven recurring JNI maintenance concern families, providing an empirical characterization of the cross-language maintenance problems that developers repeatedly address in practice.

Our empirical analysis further revealed that the existing JNI smell catalog captures only part of these maintenance concerns. While current smell definitions effectively represent several implementation-level concerns, particularly those related to resource management, exception handling, and cross-language communication, they provide only partial or no coverage for several recurring architectural and cross-language maintenance concerns. These findings demonstrate that software evolution history provides valuable complementary evidence for assessing the practical relevance and completeness of existing smell catalogs.

Based on the recurring maintenance concerns that remained inadequately represented, we proposed three candidate JNI-specific code and design smells: *Cross-Language Shotgun Surgery (CLSS)*, *Cross-Language Abstraction Leakage (CLAL)*, and *Wrong Responsibility Allocation (WRA)*. Rather than deriving these smells solely from software design principles, they are motivated by recurring maintenance evidence observed across multiple independent open-source

²<https://github.com/shahrukh-uohyd/Commit-Analysis-JSS/tree/master>

projects. For each smell, we presented its conceptual definition, representative JNI manifestation, an initial operationalization, and corresponding refactoring guidelines, providing a foundation for future automated detection techniques.

Beyond the proposed smell definitions, this work demonstrates that software evolution history constitutes an effective source of empirical evidence for validating existing smell catalogs, identifying gaps in their coverage, and motivating new smell definitions. Although the present study focuses on JNI, the proposed concern-centric methodology is independent of the underlying interoperability technology and may be applied to other multi-language ecosystems to empirically evaluate and extend their smell catalogs.

As future work, we plan to develop automated detection techniques for the proposed smells, evaluate their prevalence and impact on software quality in larger collections of multi-language software systems, investigate additional manifestations of the proposed smells, and apply the proposed methodology to other interoperability technologies beyond JNI.

CRedit authorship contribution statement

Md Shahrukh Ansari: Conceptualization, Methodology, Investigation, Formal analysis, Visualization, Writing – original draft. **Salman Abdul Moiz:** Conceptualization, Supervision, Validation, Writing – review & editing.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used ChatGPT (OpenAI) to assist with drafting and refining the manuscript and to support the qualitative analysis of JNI-related commit diffs during the maintenance concern identification process. AI-generated outputs were used as preliminary suggestions and were subsequently reviewed, manually validated, and revised by the authors. All maintenance concern labels, smell mappings, proposed smell definitions, interpretations, and conclusions were determined by the authors, who take full responsibility for the content of the published article.

References

Abidi, M., Grichi, M., Khomh, F., 2019a. Behind the scenes: developers' perception of multi-language practices, in: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, pp. 72–81.

Abidi, M., Grichi, M., Khomh, F., Guéhéneuc, Y.G., 2019b. Code smells for multi-language systems, in: Proceedings of the 24th European conference on pattern languages of programs, pp. 1–13.

Abidi, M., Khomh, F., 2020. Towards the definition of patterns and code smells for multi-language systems, in: Proceedings of the European Conference on Pattern Languages of Programs 2020, pp. 1–13.

Abidi, M., Khomh, F., Guéhéneuc, Y.G., 2019c. Anti-patterns for multi-language systems, in: Proceedings of the 24th European conference on pattern languages of programs, pp. 1–14.

Abidi, M., Rahman, M.S., Openja, M., Khomh, F., 2021. Are multi-language design smells fault-prone? an empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1–56.

Abidi, M., Rahman, M.S., Openja, M., Khomh, F., 2024. Design smells in multi-language systems and bug-proneness: a survival analysis. *Empirical Software Engineering* 29, 106.

Ansari, M.S., Moiz, S.A., 2024. Detection of unused native methods code smells in multi-language systems, in: 2024 4th International Conference on Information Communication and Software Engineering (ICICSE), pp. 44–50. doi:10.1109/ICICSE61805.2024.10625688.

Ansari, M.S., Moiz, S.A., 2025. The influence of multi-language design smells on change- and fault-proneness: An empirical study. *SSRN Electronic Journal*. URL: <https://ssrn.com/abstract=5345427>, doi:10.2139/ssrn.5345427. available at SSRN.

Cristina, M., Radu, M., Mihancea, F., et al., 2005. iplasma: an integrated platform for quality assessment of object-oriented design, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 77–80.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 2002. Refactoring: improving the design of existing code. 1999. Google Scholar Google Scholar Digital Library Digital Library.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H., 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 1–79.

Jelodar, H., Meymani, M., Razavi-Far, R., 2025. Large language models (LLMs) for source code analysis: applications, models and datasets. *arXiv preprint arXiv:2503.17502*.

Kurzyniec, D., Sunderam, V., 2001. Efficient cooperation between java and native codes—jni performance benchmark, in: The 2001 international conference on parallel and distributed processing techniques and applications.

Landis, J.R., Koch, G.G., 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 159–174. doi:10.2307/2529310.

Larman, C., 2012. Applying UML and patterns: an introduction to object oriented analysis and design and iterative development. Pearson Education India.

Li, W., Meng, N., Li, L., Cai, H., 2021. Understanding language selection in multi-language software projects on github, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE. pp. 256–257.

Liang, S., 1999. The Java native interface: programmer's guide and specification. Addison-Wesley Professional.

Mayer, P., Kirsch, M., Le, M.A., 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1–33.

Meyer, B., 1997. Object-oriented software construction. volume 2. Prentice hall Englewood Cliffs.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 1053–1058.

Winter, R.J., 2014. Agile software development: Principles, patterns, and practices: Robert c. martin with contributions by james w. newkirk and robert s. koss.

Xue, P., Wu, L., Yu, Z., Jin, Z., Yang, Z., Li, X., Yang, Z., Tan, Y., 2024. Automated commit message generation with large language models: An empirical study and beyond. *IEEE Transactions on Software Engineering* 50, 3208–3224.

Zhang, L., Zhao, J., Wang, C., Liang, P., 2024. Using large language models for commit message generation: A preliminary study, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 126–130.