Mapping Code Smells and Refactorings Accurately: Insights from an Empirical Study

Gautam Shetty, Tushar Sharma
Dalhousie University, Canada
{gautam.shetty, tushar}@dal.ca

Abstract—*Background:* **Code smells indicate underlying quality issues that negatively impact software maintainability. Refactoring is a common way to improve code quality by restructuring it, often removing these code smells. While many recommendations exist on how to refactor code smells, we do not fully understand how developers how they are removed by developers in the real world.**
Aim: **In this study, we aim to investigate the evolution of code smells and the impact of applied refactoring techniques.**
Method: **Our study addresses this gap by investigating both implementation and design smells and the refactoring techniques developers use to remove them. We also explore how often code smells are removed using established refactoring techniques. We analyzed $212,664$ commits from 87 open-source Java projects using both automated tools and manual review to understand the relationship between code smells and refactoring.**
Results: **Our key findings include: a) Extract method refactoring is most effective at fixing multiple smell types, b) Most applied refactorings do not remove code smells, c) About 82% of removed code smells are "dangling" *i.e.,* they are removed without a matching refactoring technique, and d) Design smells typically last longer in codebases than implementation smells.**
Conclusions: **This research improves our understanding of the interplay between code smells and refactoring effectiveness. Our results can help researchers develop better tools and guide software engineers in making their refactoring processes more efficient.**

I. Introduction

Software maintenance is a critical aspect of the software development lifecycle, often consuming a significant portion of development resources [1]. As software systems evolve, maintaining high code quality becomes increasingly challenging, particularly when code smells start to accumulate [2], [3]. These code smells—surface indicators of deeper design issues—increase technical debt and negatively impact software maintainability [4].

Refactoring, the process of improving the internal structure of software without altering its external behavior, has long been recognized as an effective approach to mitigating code smells and improving code quality [2]. Developers employ various refactoring techniques; among them, the common refactoring techniques include *rename*, *extract method*, and *move method*. Refactoring offers several benefits, including increased readability, reduced complexity, and, in general, enhanced maintainability [2], [5], [6].

Numerous studies have investigated tools and methodologies for examining code smells and refactoring practices. They address dimensions including detection approaches, quality impact assessment, and their relationship with software development activities and artifacts' quality [6]–[9]. Some studies attempt to understand the impact of code smells or applied refactorings on software quality attributes such as maintainability [7], [8]. Others examine how code smells evolve over time; for example, Tufano *et al.* [10] reveal that most code smells are introduced at file creation rather than through gradual changes and often persist throughout a project's lifetime. Despite these advancements, current tools mostly examine code smells and refactorings separately, limiting our understanding of how they interact and influence each other. To the best of our knowledge, the study by Yoshida *et al.* [11] is the only attempt to understand the relationship between applied refactorings and the code smells that the refactorings remove. Their short study reveals that the smells are removed by a code edition operation that does not represent a known refactoring technique. These observations indicate that there is a lack of understanding how code smells persist, evolve, or are resolved by applying specific refactoring techniques. By studying these characteristics, we can gain a deeper understanding of the connection between code smells and refactoring techniques. This knowledge can guide developers on how to make smarter choices about when and how to refactor their code, which leads to improved software maintainability.

Though existing research attempts to understand the connection between refactoring and code smells, it exhibits many limitations. First, most existing studies analyze this relationship at a coarse level and do not capture the fine-grained interactions between specific code smell instances and refactoring actions. Second, there is limited understanding of what type of refactorings to perform to resolve an existing smell. Third, prior works mainly focuses on identifying correlations between smells and refactorings as observed by automated means, without exploring the underlying semantic or meaningful relationships between them.

To address these gaps, our study conducts a detailed analysis of individual code smell instances and their corresponding refactoring actions. We also examine the lifespan and survivability of smells when analyzed at the granular level (*i.e.,* up to the scope identified by the changed lines of code) of individual smell instances. Furthermore, we investigate deeper trends that emerge from the interaction between different types of code smells and refactorings.

This study investigates the evolution of code smells over time and the impact of various refactoring techniques on smells' lifespan. We analyze $212,664$ commits gathered from 87 Java-based open-source repositories. Our analysis explores the lifecycle of code smells and examines the relationship between refactorings and smells. We provide actionable insights by mapping refactoring techniques to smells for each commit *i.e.,* identifying the refactorings used to address a type of smell, facilitating a deeper understanding of how refactoring may influence persistence of smell instances. We construct this mapping using state-of-the-art smell and refactoring detection tools, which is subsequently refined through a manual analysis. This analysis uncovered relevant connections between smells and refactorings. Additionally, we identify refactorings not associated with any smell instance to assess their role and significance in software evolution.

The primary contributions of our study are listed below.

- We present a **detailed mapping between code smell and refactoring techniques** observed in the version history of analyzed repositories. We also conduct a manual analysis of this mapping to recognize refactoring techniques that are most frequently applied and likely to be considered effective for removing particular smells.
- We analyze **dangling smells** (*i.e.,* smells that are removed without any corresponding refactoring) and **dangling refactorings** (*i.e.,* applied refactorings that do not result in removing any smell). This analysis helps us understand the inter-dependence and the gap that emerges from the lack of it.
- We perform a comprehensive **survivability analysis** of code smells over the course of a project's evolution. This reveals how long code smells typically persist.

Replication package: Our replication package including scripts and instructions to execute the scripts can be found online [12]. The collected smells, refactorings and their mapping dataset can be found here [13].

II. Methods

A. Overview

This study investigates the intricate relationship between code smells and refactoring techniques. Specifically, we *aim* to explore whether and to what extent code smells are refactored and how long they survive. Furthermore, we aim to explore refactoring techniques that do not remove smells as well as smells that are removed by changes other than commonly-known refactorings. To achieve the above goals, we define the following research questions.

RQ1. *Which refactoring techniques are used to remove individual code smells?*

This research question (RQ) focuses on understanding the mapping between code smells and corresponding refactoring techniques that are used to remove the smells. Exploring the mapping will help us better understand the concrete refactoring techniques commonly applied by software developers to address specific code smells.

RQ2. *Do refactorings always remove code smells?*

In this RQ, we examine refactorings that do not appear to remove any code smells. Additionally, we aim to investigate cases where the removal of code smells was not made by any refactoring. This analysis will help us understand the degree of inter-dependence of smells and refactorings.

RQ3. *How does the relationship between code smells and refactoring evolve over time?*

With this RQ, we examine the evolution of code smells across successive versions of a repository and understand the lifespan of smells. We aim to identify trends in the emergence, persistence, and resolution of code smells and their evolving relationship with refactoring over time.

To address the questions, we outline our approach in Figure 1. We first identify and download a set of Java repositories from GitHub. With the help of RefactoringMiner, we identify refactorings applied across all the commits of the selected repositories. Similarly, we analyze the repositories to identify code smells in their entire commit history using DesigniteJava. We develop scripts to automate these steps and analyze the collected information to establish a collation mapping between identified smells and refactoring techniques. Furthermore, we complement the automated analysis with manual assessment to ensure the correctness and reveal deeper insights. In the rest of the section, we elaborate on each of the steps.

B. Repositories selection

In step ❶ of Figure 1, we leverage the SEART GitHub search tool [14] to identify a set of repositories suitable for our research objectives. We query GitHub to fetch Java repositories that have more than $1,000$ commits, with at least $5,000$ stars, 10 contributors, and at least 100 forks excluding forked repositories. The criteria are chosen to avoid analyzing small and low-quality repositories and select repositories that show wide popularity. This criteria resulted in 183 repositories. We apply out repositories with more than $20,000$ commits to avoid excessive computing resources. We download the remaining 87 repositories using a Python script. We analyze the default branch of each repository.

C. Data preparation

Our approach requires detection of code smells and identification of refactorings throughout a repository's history *i.e.,* for each commit. The selection of appropriate tools is guided by three key requirements. First, the tools must support detection of smells and refactorings at the commit-level granularity. Second, the tools must have comprehensive detection capabilities that are well-accepted by the community and third, the tools must have an established reliability through prior validation.

Based on these criteria, we select DesigniteJava [15] to detect code smells. DesigniteJava is a state-of-the-art Java code quality analysis tool that computes various code quality metrics and supports the detection of a comprehensive set of code smells across various abstraction levels. It also supports multi-commit analysis, making it suitable for our study. This extensive coverage has led to its widespread adoption in

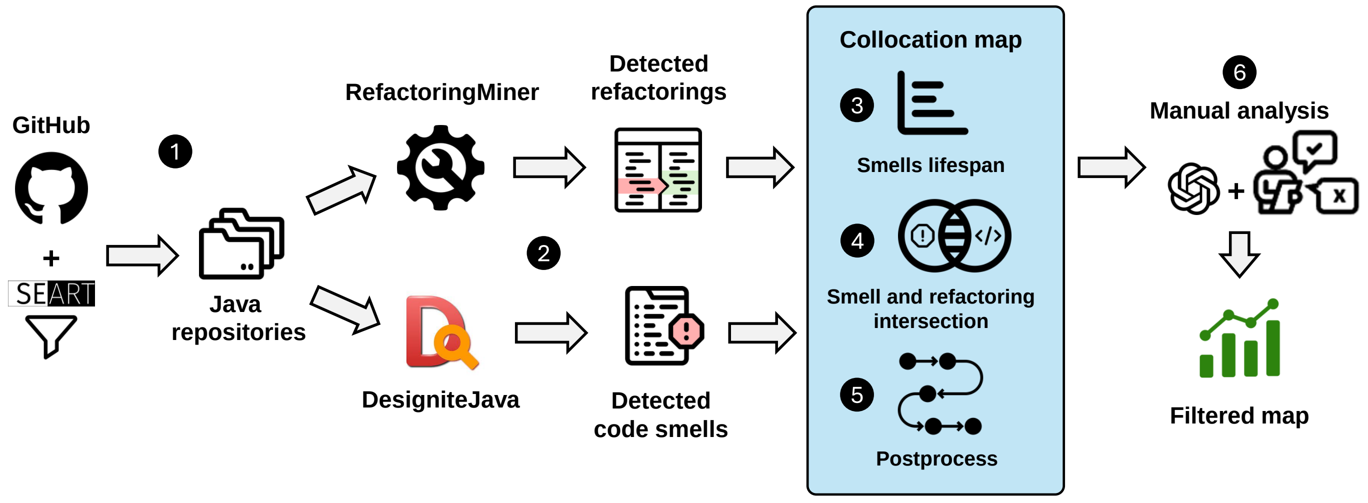


Fig. 1: Overview of the proposed approach.

numerous studies [16]–[20]. For the purposes of our study, we focus specifically on design and implementation smells to maintain a clear scope. Similarly, we select REFACTORINGMINER [21] to mine applied refactoring techniques in a Java-based repository. REFACTORINGMINER, detects over a hundred refactoring types. It has demonstrated high precision (99.6%) and recall (94%) in refactoring detection, outperforming other similar tools. It is the state-of-the-art tool widely used by software engineering community [22]–[25].

We use these tools to detect smells and applied refactorings in all the commits of the selected repositories. We use Python scripts to invoke these tools on the default branch of each repository. The detected smells and refactorings are systematically stored for further analysis.

Computational environment: We used a high-performance Linux machine equipped with 144 GB RAM and 36 cores to analyze the selected repositories. Our scripts utilize the available cores in the machine using parallelism in the form of multi-threading and multi-process tasks. We used the following key tools and development environment: DESIGNITEJAVA (v2.5.9), REFACTORINGMINER (v3.0.9), Python (v3.11.5), and Java runtime (v17).

D. Data analysis

Figure 2 describes the naming convention and different phases that characterize the life spans of smells and refactorings used in this study. The figure shows evolution of a repository in terms of commits, detected smells, and refactoring horizontally. At phase a, a new $smellInstance$ is *introduced* at commit c_2. In subsequent commits, the location of a $smellInstance$ may change, or the $smellInstance$ may get removed.

We show a $smellInstance$ at b that is moved locally due to changes in the code (for example, when a few lines are deleted in the same file before the smell location). It is also possible that the smells are *moved* to another file (for example, due to renaming the file). In this scenario, other refactoring actions that do not directly affect the smell are taken into consideration, and $smellInstance$ metadata is updated accordingly (if there is any). So when this step is performed on $smellInstance$, it is termed as $movedSmell$.

In step c, at commit c_{k+1}, the $smellInstance$ is no longer present compared to the previous commit; in this case, the smell is considered removed. As this $smellInstance$ does not persist after commit c_k, it is termed as $removedSmell$. Here, the refactorings' right side diff range is checked for intersection with any $smellInstances$' range at the same commit. If they intersect, the particular refactoring is mapped to $smellInstance$.

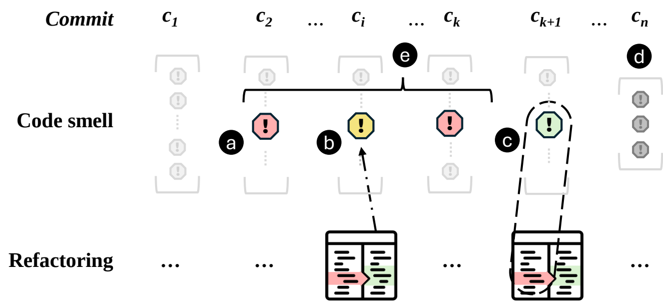


Fig. 2: Naming convention and smell lifespan

We calculate the lifespan of a $removedSmell$, in terms of number of $commits$ and number of $days$, it took to remove a particular $smellInstance$. For example, the difference between c_k and c_2, as indicated by e, is the lifespan of the smell. During this process at the latest version of the repository, not all $smellInstance$ are considered $removedSmell$. Some smells, such as shown at d, remain; these are a set of $aliveSmells$. These smells are $smellInstances$, which still persist in the latest version of the repository. We do not compute lifespan for $aliveSmells$ since those are still active and do not have an end $date$ or an end $commit$.

1) Smell and refactoring mapping: Identifying collocated refactoring changes and code smells is an important step of our analysis. We map a code smell with an applied refactoring, if the scope of the both the smell and refactoring is same and the smell is removed in the commit due to the applied refactoring. The scope of a code smell or refactoring is defined by its file location and the specific lines of code it encompasses.

For *implementation smells*, we identify the commit where a smell is removed and analyze the original code (*i.e.,* left-side of the code diff) as shown in Figure 2 for the presence of refactorings. We then compare the range, in terms of lines of code, of the refactoring operation with the range of the removed smells.

We also map refactorings that introduced smells by considering the changed code (*i.e.,* the right-side of code diff). We only consider changes where the range of refactorings intersects with the smells' range for mapping. To summarize, let S be the set of smell instances and R be the set of refactoring instances. For each $s \in S$ and $r \in R$, we define a mapping function M:

$$M(s,r) = \begin{cases} 1, & \text{if } range(s) \cap range(r_{\text{left}}) \neq \emptyset \\ & \text{(for smell removal)} \\ 1, & \text{if } range(s) \cap range(r_{\text{right}}) \neq \emptyset \\ & \text{(for smell introduction)} \\ 0, & \text{otherwise} \end{cases}$$

where $range(s)$ represents the method's range in lines of code affected by smell s, $range(r_{\text{left}})$ and $range(r_{\text{right}})$ represent the range of refactorings found in the left-side or right-side of the code-diff.

For *Design smells* [8], we identify the newly introduced and removed smells and applied refactorings in a commit. We map smells to refactorings based on their common scope. We consider the scope of a smell and refactoring same when the refactoring is applied in the same file(s) where the smell is detected. Let $F(s)$ and $F(r)$ be the set of files affected by a smell s and a refactoring r in a commit respectively. The mapping function for the smell s and the refactoring r is defined as:

$$M(s,r) = \begin{cases} 1, & \text{if } F(s) \cap F(r) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

2) Smells lifespan calculation: We determine a code smell's lifespan by counting the commits between its introduction and removal. Given that many smells may occur multiple times in a code block (such as *magic number*) and they may move within and outside the file, the lifespan computation is complex. We develop an algorithm that tracks each smell instance from its inception to its resolution considering complex cases. Algorithm 1 presents the smell lifespan calculation mechanism. In the algorithm, c, p, and t refer to current commit, previous commit, and commit timestamp, respectively.

Algorithm 1: Smell lifespan calculation

```
Input: Commits sorted (by time in ascending order)
       $C = \{(c_1,t_1),(c_2,t_2),...,(c_n,t_n)\}$
Input: Smell instances $S_c$ detected for each commit $c$
Output: List of smell instances across repository history
1  $liveSmells \leftarrow \emptyset$; $smellInstances \leftarrow \emptyset$;
2  foreach $(c,t) \in C$ do
3      $prevSmells \leftarrow S_p$;
4      $currSmells \leftarrow S_c$;
5      foreach $s \in currSmells \cap prevSmells$ do
6          if $s$ is "implementation smell" then
7              $liveSmells[s].range \leftarrow s.range$;
8              $liveSmells[s].commit \leftarrow (c,t)$;
9          end
10     end
11     $addedSmells \leftarrow currSmells - prevSmells$;
12     foreach $s \in addedSmells$ do
13         $liveSmells[s].introducedCommit \leftarrow (c,t)$;
14     end
15     $liveSmells \leftarrow liveSmells \cup addedSmells$;
16     $removedSmells \leftarrow prevSmells - currSmells$;
17     foreach $s \in removedSmells$ do
18         $liveSmells[s].removedCommit \leftarrow (c,t)$;
19         $smellInstances \leftarrow$
             $smellInstances + \{liveSmells[s]\}$;
20     end
21     $liveSmells \leftarrow liveSmells - removedSmells$;
22     $p \leftarrow c$;
23 end
   // Handle smells that were never removed
24 foreach $s \in liveSmells$ do
25     $smellInstances \leftarrow smellInstances + \{s\}$;
26 end
27 return $smellInstances$;
```

The algorithm processes a repository's commit history in ascending order of commits, maintaining a set of active smells (*liveSmells*) and a list of all smell instances (*smellInstances*). For each commit, it identifies *addedSmells* as those present in the current commit but not in the previous one and adds them to *liveSmells*. It detects *removedSmells* as those present in the previous commit but not in the current one, marking their resolution and recording their lifespan in *smellInstances*. For moved smells (specific to implementation smells), the algorithm tracks changes in line numbers without considering them as removed, updating their metadata in *liveSmells*. Finally, any smells remaining in *liveSmells* after processing all commits are added to *smellInstances*, ensuring all lifespans are captured including smells that were never removed. The algorithm outputs a list of smell instances, each representing the lifespan of a smell in terms of smell metadata, commit count, and temporal duration.

Smell Lifespan in Commits (SLC$_s$) for a smell instance s is calculated as the number of commits from its introduction to its removal. For example, if a smell is introduced in commit c_i and removed in commit c_j, then SLC$_s$ is $j-i+1$. This metric quantifies the duration, in terms of the number of commits, taken to address a smell. We also compute *Smell Lifespan in Time* (SLT$_s$), which measures the time (typically in number of days) between the introduction and removal of a smell s.

E. Postprocessing

We carry out a mapping between code smells and identified refactoring operations in Section II-D1. However, across multiple commits in a repository, there can be several smell instances that are semantically identical but appear as separate instances due to minor changes in metadata, such as variations in package name, type name, or method name of the smell. For instance, if a method `foo` that suffers from a complex method code smell, is renamed to `bar`, then `bar` should not be considered as a new instance of complex method". These variations are often caused due to move or rename type refactorings, which result in the creation of new smell instances instead of updating the existing ones.

To address this issue, we introduce a postprocessing step, indicated as step 5 in Figure 1. This step links smell instances that were split due to refactoring operations. Specifically, we identify cases where a refactoring removes one smell instance and introduces another, and we merge these as a single instance specially when the old and new smell instance are essentially the same instance but differs in a minor way (such as change in the starting line or a change in the type name). Through this process, we identify a total of $74,450$ smell instances initially terms as different but were merged after analyzing the occurrence pattern and associated metadata.

F. Manual analysis

After refining the initial mapping by aligning code smells with refactorings, some associations remained practically invalid. This limitation arises because the mapping is primarily based on collocation, which does not inherently guarantee meaningful relationships between code smells and refactoring actions. In a similar study, Yoshida *et al.* [11] also observed the spurious mapping between code smells and refactoring when the basis of mapping is collocation. To assess the correctness of the identified associations, a thorough manual verification is essential. The aim of the manual analysis is to identify and eliminate spurious smell-refactoring mappings and improve the reliability of the generated dataset.

For this manual analysis, we select the five most frequently applied refactoring techniques corresponding to each identified code smell in the dataset. For each of these top smell-refactoring pairs, we randomly select six instances, resulting in a total of 802 samples for manual evaluation. The analysis is conducted by the first author of the study primarily, who has 3 years of software development experience. The assessor analyze the collected metadata, including the mapped smell and refactoring names and changes in the code to assess whether the mapped smell and refactoring pair is genuine and must be kept. For example, class $javac.handlers.HandleBuilder$ in this commit [26] is genuine, as refactoring technique *Change method access modifier* help resolve smell *Imperative abstraction* in this class. Whereas, class $ApiAccessLogRespVO$ [27] has refactoring *Remove class annotation* mapped to removal of smell *Unnecessary abstraction* because both affects the same class. But the refactoring practically never impacts the smell, hence these types of pairs were considered impractical and discarded in further study.

The assessor also employ a Large Language Model (LLM) to avoid bias in the assessment. We use state-of-the-art model from OpenAI *GPT-4o-mini-2024-07-18* for the task. We design a suitable prompt for the task based on the recommended best practices [28]. We conduct a short pilot study to ensure that the prompt is designed appropriately and providing reasonable response for the given samples. The prompt that we used is listed below.

Role: You are an expert software developer with a strong understanding of code smells and refactoring, especially in Java programming.
Instructions:

- $correct\_mapping$: Respond with $true$ if the refactoring effectively removes the code smell; respond with $false$ if it does not, and provide a brief explanation.
- **Output Format:** The answer must be in the format: $\{correct\_mapping : true/false,$ $reason : < briefexplanation >\}$

Context: A code smell was removed in a commit with the following details:

- **File:** $smell\_file\_name$
- **Method:** $smell\_method$ (Lines s_method_range)
- **Smell Kind:** $smell\_kind$
- **Smell Type:** $smell\_type$

A refactoring was applied in the same commit:

- **Refactoring Type:** ref_type
- **Description:** $ref\_description$
- **Code Changes:**
  - **Before:** $ref\_left\_changes$
  - **After:** $ref\_right\_changes$

We invoke the LLM for each of the 802 samples for which we already have an assessment from one human assessor. We compare the responses provided by the LLM with the assessor's assessments and flag the samples where the LLM is in disagreement with the assessor. These flagged samples were further reviewed by another author to ensure accuracy. The inter-rater agreement between the human assessor and the LLM responses is measured using Cohen's Kappa ($\kappa = 0.84$).

One common observation after human assessment was the coincidental collocation of smells. A smell-refactoring pair is considered spurious or coincidental when the pair is collocated; however, the refactoring semantically does not and, sometimes, cannot remove the smell occurrence. Instead, the refactoring modifies other aspects, such as a shared method or class associated with the smell. During the manual analysis, we find such coincidental collocation in 115 (out of 135) smell-refactoring pairs.

For example, *long statement* is mapped to *add parameter* refactoring based on collocation analysis in this commit [29] . In $propagatedHeaders$ method in type $Protocol$ belonging to package $org.asynchttpclient.providers.netty4.handler$, we have a *long statement* at line 116. This smell was removed in this commit, as visible in the left side diff of the commit.

Add parameter refactoring is mapped to this smell. A new parameter $boolean\, switchToGet$ is added to the same method $propagatedHeaders$ of smell does affect the smell's removal. The reason the refactoring is mapped to the $smellInstnace$ lies in the limited factual information, *i.e.,* intersection in method range in this scenario. Semantically, the refactoring does not impact smell because *add parameter* refactoring only adds a new parameter to the method definition that do not affect the smelly statement length in any way. Hence, such kind of smell-refactoring pair is considered incorrect and excluded from set of filtered smell-refactoring pairs.

Additionally, we also observe that some of the design smells, such as *wide hierarchy*, *unutilized abstraction*, *rebellious hierarchy*, *multipath hierarchy*, *hub-like modularization*, *cyclically-dependent modularization*, *cyclic hierarchy*, and *broken hierarchy*, cannot be mapped with the adopted collocation approach. Their accurate mapping requires considerations such as the dependency graph and selective code change mapping to refactorings.

This evaluation process identify spurious smell-refactoring pairs arising merely due to coincidental collocations. By removing these spurious mappings, we ensure that our final dataset contains only correct and consistent smell-refactoring pairs. This curated mapping serves as a reliable basis for further analysis in our study.

III. Results

A. RQ1: Smell and refactoring mapping and understanding their relationship

Approach: RQ1 aims to identify the common refactoring techniques applied to resolve code smells. To achieve this objective, we map the identified smells to respective refactorings using state-of-the-art tools DesigniteJava and RefactoringMiner, as discussed in Section II-D1. We ensure that minor changes in the code do not result in a new set of smells by following a set of postprocessing steps (Section II-E). To further verify the mappings and reduce the coincidental and spurious collocations, we conduct a thorough manual analysis. We outline the steps involved in manual analysis in II-F. Manual analysis helps us provide meaningful pairs of smells and refactorings in the generated mapping. Therefore, result of the manual analysis is a set of smell-refactoring mappings that needs to be discarded from the final analysis. We refer to our dataset as the filtered dataset after filtering out such mappings; the retained refactoring techniques corresponding to each smell is referred as the *effective* refactoring technique.

Results: Figure 3 illustrates the identified mapping between smells and refactorings. Our observations show that 15.1% detected smells remain alive without significant changes. The majority (71.1%) of detected smells are not mapped to any refactorings that may help resolve the smell. Only a fraction of detected removed smells (13.8%) are mapped to a refactoring technique.

Figure 4 visualizes the evolution of code smells across multiple software repositories over normalized development time. The X-axis represents the commit history, normalized from 0

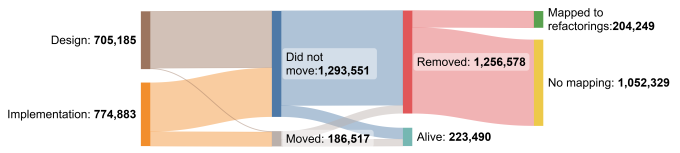


Fig. 3: Code smell-refactoring mapping overview

(first commit) to 1 (last commit), which allows for consistent aggregation across projects with varying commit counts. The Y-axis shows the average smell count per repository and is displayed on a logarithmic scale for better readability.

The *Smells Introduced per Commit* curve, shown in green, represents the average number of new code smells introduced at each normalized point in time. The *Smells Removed per Commit* curve, shown in red, represents the average number of smells removed at each point. Removals are constrained to only eliminate smells that were previously introduced. The *Net Alive Smells* curve, shown in blue, reflects the number of active smells remaining in the codebase over time. This is calculated by tracking the net change between introduced and removed smells. The *Cumulative Smells Removed* curve, shown as an orange dashed line, indicates the total number of smells removed up to each point in time.

The aggregated time series shows the cumulative number of removed smells often exceeds the count of alive smells by the end of the timeline. This occurs because smells can be introduced and removed repeatedly across different commits, leading to a higher overall count of removed smells than those that persist.

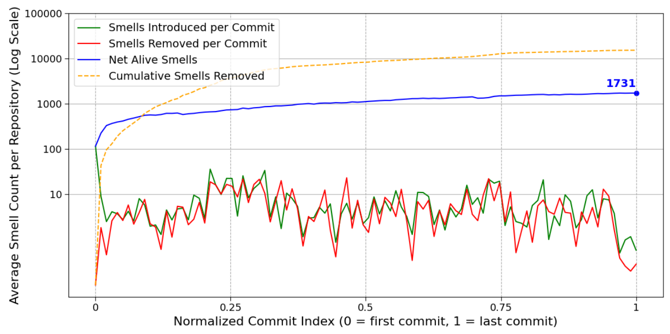
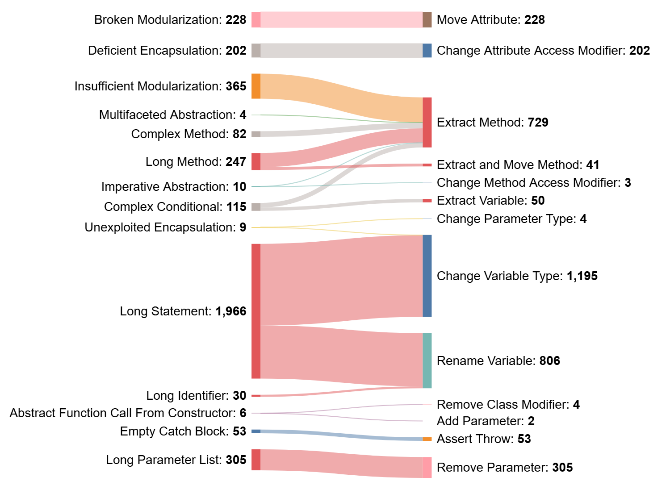


Fig. 4: Code smell evolution

Figure 5 presents the mapping of code smells and corresponding Refactoring techniques that we find effective for smell resolution. Our results indicate that the *extract method* refactoring is the most frequently applied technique for addressing multiple types of code smells. It is also the most effective in resolving various types of smells, including *long method*, *complex method*, *complex conditional*, *multifaceted abstraction*, *imperative abstraction* and *insufficient modularization*. One example of *extract method* refactoring to eliminate an implementation smell, specifically *long method*, can be observed in the function *writeField* in type *Structure* from

$1ae7d$ commit of "*java-native-access/jna*" repository [30].



Broken Modularization: 228
Deficient Encapsulation: 202
Insufficient Modularization: 365
Multifaceted Abstraction: 4
Complex Method: 82
Long Method: 247
Imperative Abstraction: 10
Complex Conditional: 115
Unexploited Encapsulation: 9
Long Statement: 1,966
Long Identifier: 30
Abstract Function Call From Constructor: 6
Empty Catch Block: 53
Long Parameter List: 305
Move Attribute: 228
Change Attribute Access Modifier: 202
Extract Method: 729
Extract and Move Method: 41
Change Method Access Modifier: 3
Extract Variable: 50
Change Parameter Type: 4
Change Variable Type: 1,195
Rename Variable: 806
Remove Class Modifier: 4
Add Parameter: 2
Assert Throw: 53
Remove Parameter: 305

Fig. 5: The refactoring techniques used to refactor each type of code smells

Another interesting case is the *remove parameter* refactoring, which directly addresses and reduces the severity of the *long parameter list* smell. Other notable refactorings include *change method access modifier*, which is effective in removing design smell *imperative abstraction* and *assert throw* refactoring to resolve *empty catch block*.

Some refactorings did not completely resolve smells most of the time, but they help reduce the severity of smells. For example, in commit $e90736$ of project "*cryptomator*" [31] , method $FxApplication$ (line 64), *Remove parameter* refactoring helps reduce the severity of smell type *Long parameter list*. Although this refactoring did not resolve the smell, but helped reduce the severity of the smell.

RQ1 Summary. We identify smell-refactoring pairs where specific refactoring techniques effectively remove code smells. *Extract method* refactoring is the most effective in resolving multiple types of smells.

B. RQ2: Unmapped smells and refactorings

Approach: RQ2 aims to investigate the *dangling* links in the code smell and refactoring mapping analysis. A code smell instance is referred to as dangling when removal of the smell is not mapped with any refactoring technique. Similarly, an applied refactoring is treated as a dangling refactoring instance when it does not remove a smell. Our collocation logic that maps detected code smells and applied refactorings in each commit of a repository reveals such unmapped smells and refactorings. We elaborate the collocation mechanism in Section II-D.

We also analyze such dangling links to understand the reasons behind their occurrence. We randomly select ten samples from each dangling smell type. We analyze each instance by manually examining the code that introduces the smell, the intermediate commits that retain it and changes in the code that removes the smell. This verification is carried out using the raw smell data produced by DESIGNITEJAVA as well as version control metadata and corresponding code changes. When we identify a refactoring in the code that successfully removes a code smell, we analyze the raw data generated by REFACTORINGMINER to investigate why the tool failed to detect this refactoring. Similarly, we analyze dangling refactorings pairs to understand the reason behind such unmapped refactorings. Here, we randomly select five samples for each refactoring type in the top 15 dangling refactoring types identified across the corpus.

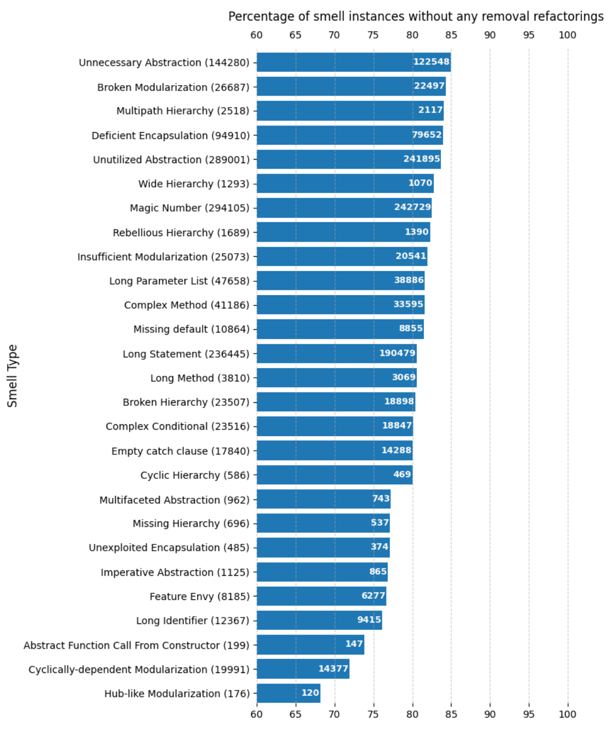


Fig. 6: Distribution of dangling code smells

Results: Figure 6 shows dangling code smells *i.e.,* the number of code smell instances where no refactorings are identified against their removal. About 82% of smell instances do not have any removal refactorings mapped. We further analyze the dangling smells to identify their root cause. We manually analyze 270 such smells and categorize our findings in the following categories.

Valid dangling smells: These are genuine smell removals, but the mapping to refactorings is partially or completely missing.

- *Removal without any refactorings involved* (48.7%): Smells are removed due to general code changes not classified as standard refactorings.
- *Removal with refactoring involved, but missed correct mapping* (0.6%): Smells are removed via valid refactorings and the refactoring is detected; however, the mapping of the refactoring to smells is missed.
- *Removal with refactoring involved, but tool missed refactoring identification* (5.6%): Smells are removed by refac-

torings, but RefactoringMiner failed to detect these refactorings, leading to unmapped $smellInstances$.

Invalid dangling smells: These are cases where the smell appears to be removed due to other reasons.

- *Version workflow issue* (20.2%): The $smellInstances$ removed due to complex Git workflows, such as merges from older commits or squash operations during rebase. These scenarios obscure the actual timeline of changes, leading to a false impression of smell removal.
- *Needs further analysis* (24.9%): Some design smells need deeper semantic or dependency analysis, especially when analysis of a smell instance is dependent on the type dependency graph.

Similarly, we also observe that half of the identified refactorings (51.82%) are not mapped to any smell instances. Table I shows the top 15 dangling refactoring types. To better understand the persistence of these dangling refactoring instances, we manually analyze 75 such refactoring instances and categorize the causes in the following categories:

Valid dangling refactorings: These are valid refactoring operations, but their mapping to smell removals is either absent or incomplete.

- *Refactoring instances without any smells involved* (84.57%): Refactorings in this category were applied independently and did not lead to removal of any supported code smell.
- *Refactoring instances with smells involved, but missed correct mapping* (0.2%): The related smell instance is present and removed, but the mapping between the smell and the refactoring is not established.
- *Refactoring instances with smells involved, but the tool missed smell identification* (0.4%): Although the refactoring removed a relevant smell, DesigniteJava did not identify its presence.

Potentially invalid dangling refactorings (14.83%)**:** A commit has several changes, and if refactoring-related changes are less than half of the changed lines of code, it indicates that the refactorings are incidental and potentially side effects of other changes.

TABLE I: Top 15 dangling refactoring types

Refactoring technique	Count	Distribution
Add method annotation	50,517	8.58%
Change variable type	37,447	6.36%
Change return type	33,692	5.72%
Change parameter type	33,605	5.71%
Add parameter	30,519	5.18%
Rename method	22,104	3.76%
Change method access modifier	21,907	3.72%
Rename parameter	20,247	3.44%
Change attribute type	19,022	3.23%
Rename variable	15,882	2.70%
Extract method	15,498	2.63%
Extract variable	14,446	2.45%
Remove method annotation	14,386	2.44%
Remove parameter	14,243	2.42%
Move class	13,310	2.26%
other refactoring types	231,789	39.38%

RQ2 Summary. Our results show that the majority of removed code smells (~82%) are dangling smells, primarily because they are removed through normal feature changes or bug fixes rather than explicit refactorings. Additionally, over half of the detected refactorings (~ 52%) are dangling refactorings, often performed independently during routine maintenance. The findings suggest that many smell removals and refactorings occur incidentally during general development activities rather than intentional cleanup.

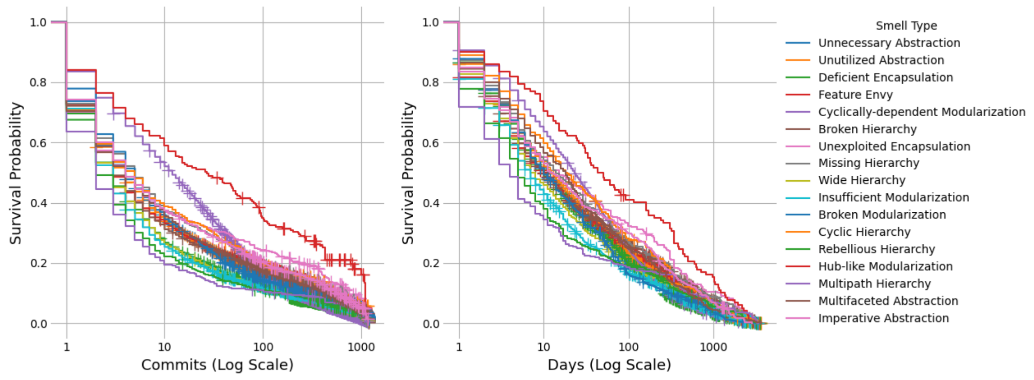
C. RQ3: Smells lifespan analysis

Approach: The objective of RQ3 is to examine how code smells evolve over time and how long they persist. To answer this question, we first calculate the lifespan of smell instances using the approach described in Section II-D2. We measure the persistence of each smell instance in terms of the number of commits it spans and its duration in days. Based on this data, we perform survival analysis using Kaplan-Meier curves [32] to assess the persistence patterns of different types of smells.

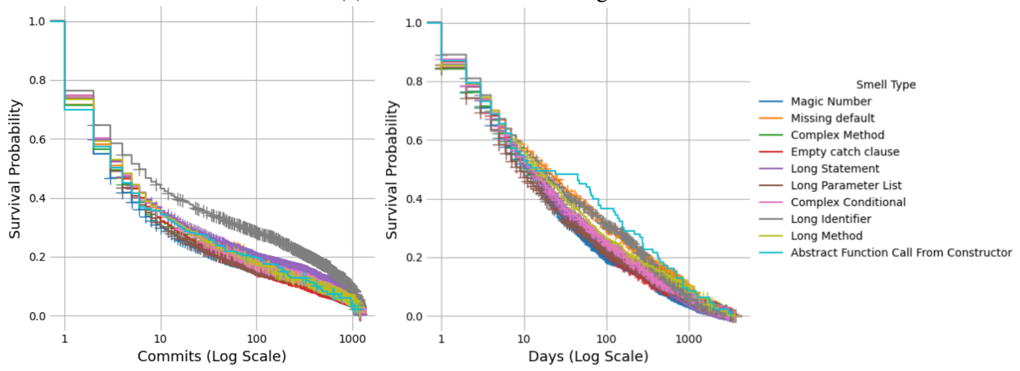
TABLE II: Survival probabilities for code smells across all projects for both commits and days over 10 and 100 range. Values highlighted in green refer to low survival probability, while those in red indicate high survival probability.

Smell Type	Survival Probability in commits		Survival Probability in days	
	10	100	10	100
Broken hierarchy	0.31	0.18	0.50	0.26
Broken modularization	0.35	0.14	0.49	0.16
Cyclic hierarchy	0.39	0.20	**0.60**	0.28
Cyclically-dependent modularization	**0.52**	0.15	**0.64**	0.24
Deficient encapsulation	0.27	**0.12**	0.49	0.20
Feature envy	0.33	0.19	0.50	0.25
Hub-like modularization	**0.59**	**0.35**	**0.70**	**0.41**
Imperative abstraction	0.37	0.20	0.53	0.30
Insufficient modularization	0.25	**0.12**	0.43	0.17
Missing hierarchy	0.38	0.20	0.54	0.27
Multifaceted abstraction	0.36	0.17	0.56	0.25
Multipath hierarchy	**0.19**	**0.11**	0.34	0.18
Rebellious hierarchy	0.22	**0.11**	0.38	0.19
Unexploited encapsulation	0.37	0.24	0.55	0.32
Unnecessary abstraction	0.34	0.16	0.52	0.22
Unutilized abstraction	0.34	0.16	0.52	0.23
Wide hierarchy	0.26	**0.12**	0.46	0.18
Abstract function call from constructor	0.34	0.20	0.53	**0.37**
Complex conditional	0.34	0.17	0.52	0.25
Complex method	0.32	0.16	0.52	0.24
Empty catch clause	0.32	0.16	0.52	0.20
Long identifier	**0.43**	**0.28**	**0.56**	0.31
Long method	0.34	0.18	**0.56**	0.26
Long parameter list	0.30	0.16	0.47	0.22
Long statement	0.36	0.19	0.53	0.23
Magic number	0.30	0.16	0.51	0.20
Missing default	0.35	0.19	**0.57**	0.30

Results: Table II presents the survival probabilities of various smells measured in both commits and days. On average, design smells have a survival probability of 0.35 after 10 commits and 0.19 after 100 commits. In contrast, implementation smells average 0.33 and 0.17 for the same thresholds,



(a) Survival curve for design smells



(b) Survival curve for implementation smells

Fig. 7: Code smell survival curves based on survivability of smell by commits and days

respectively. This suggests that design smells generally persist longer in the codebase than implementation smells. Smells such as *cyclic-dependent modularization* and *hub-like modularization* show high survival probabilities, indicating they are less likely to be removed promptly. On the other hand, smells like *multipath hierarchy* and *rebellious hierarchy* have lower survival rates, implying they are resolved more quickly. Notably, while *cyclic-dependent modularization* persists in the short term (10 commits), it is more likely to be addressed by the 100^{th} commit.

Among implementation smells, *long identifier* exhibits the highest survival probabilities. Most other implementation smells show similar trends, with approximately 0.33 survival at 10 commits and 0.17 at 100 commits. These findings suggest that developers may prioritize fixing certain types of smells over others, potentially due to their perceived impact on maintainability or system behavior. The differences in survival patterns between design and implementation smells also highlight the need for targeted refactoring strategies tailored to the nature and persistence of each smell type.

Figure 7 shows the Kaplan-Meier survival curves for both design and implementation smells. Most smell instances do not survive beyond 10 commits. Design smells generally exhibit longer lifespans than implementation smells. Consistent with Table II, smells such as *hub-like modularization*, *cyclic-dependent modularization*, and *long identifier* demonstrate strong resistance to removal. An interesting case is *abstract function call from constructor*, which shows higher survivability in terms of time (days), even if not as pronounced in commit count.

RQ3 Summary. Among the smells that are eventually removed, most types are typically resolved within 100 commits from their introduction. However, design smells tend to have a higher survival probability than implementation smells across both commit and time-based survival analysis. Our results also indicate that certain smells remain more resistant to removal, emphasizing the importance of targeted refactoring strategies to address more persistent smells.

IV. Related Work

A. Smell and refactoring detection

Several studies examine code smell occurrences and applicable refactorings. Shahidi *et al.* [33] proposed an automated approach to identify and refactor long-method smells using the extract method refactoring for Java code. They used cohesion metrics and advanced graph analysis techniques to identify extract method opportunities. De Stefano *et al.* [34] developed an IntelliJ IDEA plugin for automated detection and refactoring of common types of code smells.

There have been studies that focus on the intersection of code smells and refactorings. For example, Lacerda *et al.* conducted a tertiary systematic review on code smells and refactoring, highlighting their relationship with software quality attributes such as maintainability and testability [7]. They identified key research items including smell detection techniques, refactoring approaches, and empirical studies on smell impact and trends. Also, Yoshida *et al.* revisited the relationship between code smells and refactoring [11]. Their study investigated whether developers apply appropriate refactoring patterns to fix code smells in three open-source software systems. This work bridges the gap between Fowler's catalog of refactoring patterns and empirical observations, providing insights into the practical application of refactoring to address code smells.

B. *Code smell evolution*

Analyzing temporal dimension reveal insightful observations related to characteristics of code smells and their resolution. Tufano *et al.* conducted a study on when and why code starts to smell bad [10]. They analyzed the evolution of bad smells in 200 open-source projects, finding that most bad smells are introduced when files are created and not necessarily due to decay. Their work provides insights into the lifecycle of code smells and the factors contributing to their introduction. Similarly, Chatzigeorgiou and Manakos investigated the evolution of code smells in object-oriented systems [35]. Their study on open-source projects revealed that the majority of code smells persist throughout the project's lifetime, and very few smells are removed from a system after their introduction. They also found that changes that introduce smell often have a structural nature, such as the addition of methods or classes.

At a high abstraction level, Gnoyke *et al.* analyzed architecture smell in 14 open source systems in 485 versions, exploring their role in system degradation [36]. The study identifies that architecture smells remain stable relative to code size, with certain types, such as cyclic dependencies, having a more significant impact on degradation. These insights help practitioners address system degeneration and guide researchers in managing architecture smells and technical debt. Similarly, Sas *et al.* investigated the relationship between architectural smells and changes in source code [37]. They found that certain types of code changes are more likely to introduce or remove architectural smells. Kim conducted an empirical study specifically on the evolution of test smells [38]. The study analyzed how test smells are introduced and evolve over time in open-source projects. The results showed that certain test smell types tend to persist longer than others.

However, there is a lack of detailed and thorough studies considering a comprehensive set of smells and refactoring carefully examining the relationship between refactorings and smells. Such relationship mapping between code smells and refactoring requires significant human expertise that are not by default embedded in automated tools. We fill this research gap by first setting a detailed approach to identify and map smells and refactorings by using automated tools and scripts but also analyzing the identified mappings manually to reason with and validate the identified relationships. Such approach provides a deeper understanding of how refactorings influence the persistence of smells.

V. Threats to validity

External validity: This study focuses on software projects written in Java, primarily because of the extensive research literature available on Java code quality and the robust tools for analyzing Java codebases. To enhance the generalizability of our findings, we initially selected 183 popular open-source Java repositories based on criteria ensuring their maturity and quality. However, due to the computational demands and time constraints of analyzing every commit, we limited the final selection to 87 repositories included in this study.

Internal validity: A potential threat to internal validity arises from the manual human analysis used to validate smell-refactoring pairs. To mitigate this, we employed LLM-based validation through prompt engineering. Additionally, a third researcher resolved any conflicting cases.

Construct validity: Construct validity concerns whether the tools and metrics used truly represent the phenomena being studied. Our study relies on two external tools: DesigniteJava for detecting code smells and RefactoringMiner for identifying refactorings. Incorrect detection from either tool poses a risk to the accuracy of our results. Though these tools show some limitations, they are still the best available tools for the smell and refactoring detections tasks, respectively. To minimize the issues originating from tools, we manually reviewed issues and discuss the limitations in the paper to open the door for fixing those issues in the tools in the future.

VI. Conclusions and future work

This study explored design and implementation smells with mapped refactorings by leveraging experimental evidence gathered via a large-scale empirical study on 87 Java repositories. Specifically, the study analyzed whether certain refactorings are more commonly applied to remove specific types of code smells (RQ1). Our manual analysis ensured the correctness of the identified mappings between smells and refactorings. We also analyzed the unmapped smells and unmapped refactorings to explore possible reasons for the lack of mapping (RQ2). Additionally, our survivability analysis revealed which smells persist longer during repository development. The findings show that design smells tend to survive longer than implementation smells. In particular, *unexploited encapsulation* and *missing hierarchy* exhibited long survival times, whereas *missing default* had the shortest survival time (RQ3).

In the future, we would like to address the remaining complexities and corner cases in improving the mapping between smells and refactoring. We also would like to explore other kinds of smells such as test and architecture smells and impact of refactorings on them.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[5] W. F. Opdyke, "Refactoring: A program restructuring aid in designing object-oriented application frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[6] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[7] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[8] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[9] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450–477, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218301444

[10] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[11] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–4.

[12] G. Shetty and T. Sharma, "Code Smell Evolution." [Online]. Available: https://github.com/SMART-Dal/code_smell_evolution

[13] ——, "Dataset for empirical study: Code smells and refactorings," Apr. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15285379

[14] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[15] T. Sharma, "Multi-faceted code smell detection at scale using designitejava 2.0," in *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*. ACM, 2024, pp. 284–288. [Online]. Available: https://doi.org/10.1145/3643991.3644881

[16] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 346–357.

[17] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering*, vol. 25, pp. 4020–4068, 2020.

[18] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of design problems through refactorings: Are we looking at the right symptoms?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 148–153.

[19] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 511–522.

[20] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, p. 12, 2018.

[21] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[22] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–48, 2023.

[23] M. T. Hasan, N. Tsantalis, and P. Alikhanifard, "Refactoring-aware block tracking in commit history," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3330–3350, 2024.

[24] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv preprint arXiv:2110.09610*, 2021.

[25] P. Alikhanifard and N. Tsantalis, "A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–63, 2025.

[26] Lombok, "added package-infos to lots..." https://github.com/projectlombok/lombok/commit/09ea02e4f5752e615be2ff5177be1fb328702a5b, 2014.

[27] YunaiV, "Simplify errorlog and accesslog modules v0," https://github.com/yunaiv/yudao-cloud/commit/930cdce7a0e0af87e1621150a32fdb431a1a49b2, 2023.

[28] OpenAI. [Online]. Available: https://platform.openai.com/docs/guides/text?api-mode=responses

[29] AsyncHttpClient, "Propagate most headers on redirect, close #824," https://github.com/asynchttpclient/async-http-client/commit/f46ed2fb5674778c9836fc8a3e965cbaa7b07cae, 2015.

[30] Java-Native-Access, "Consolidate structure ffi type info..." https://github.com/java-native-access/jna/commit/1ae7d8373f9fcaf403897a33d6ea8536602b59fa, 2017.

[31] Cryptomator, "refactored launcher, deleted uilaunchermodule," https://github.com/cryptomator/cryptomator/commit/e9073604193b18b59ae3122c489a7fba8e1e0452, 2022.

[32] W. N. Dudley, R. Wickham, and N. Coombs, "An introduction to survival statistics: Kaplan-meier analysis," *Journal of the advanced practitioner in oncology*, vol. 7, no. 1, p. 91, 2016.

[33] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, vol. 187, p. 111221, 2022.

[34] M. De Stefano, M. S. Gambardella, F. Pecorelli, F. Palomba, and A. De Lucia, "casper: A plug-in for automated code smell detection and refactoring," in *Proceedings of the International Conference on Advanced Visual Interfaces*, 2020, pp. 1–3.

[35] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, pp. 3–18, 2014.

[36] P. Gnoyke, S. Schulze, and J. Krüger, "An evolutionary analysis of software-architecture smells," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 413–424.

[37] D. Sas, P. Avgeriou, I. Pigazzini, and F. Arcelli Fontana, "On the relation between architectural smells and source code changes," *Journal of Software: Evolution and Process*, vol. 34, no. 1, p. e2398, 2022.

[38] D. J. Kim, "An empirical study on the evolution of test smell," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 149–151.