Towards Understanding Refactoring Engine Bugs

HAIBO WANG, Concordia University, Canada
ZHUOLIN XU, Concordia University, Canada
HUAIEN ZHANG*, Hong Kong Polytechnic University, China
NIKOLAOS TSANTALIS, Concordia University, Canada
SHIN HWEI TAN, Concordia University, Canada

Refactoring is a critical process in software development, aiming at improving the internal structure of code while preserving its external behavior. Refactoring engines are integral components of modern Integrated Development Environments (IDEs) and can automate or semi-automate this process to enhance code readability, reduce complexity, and improve the maintainability of software products. Like traditional software systems, refactoring engines can generate incorrect refactored programs, resulting in unexpected behaviors. In this paper, we present the first systematic study of refactoring engine bugs by analyzing bugs arising in three popular refactoring engines (i.e., Eclipse, IntelliJ IDEA, and Netbeans). We analyzed these bugs according to their refactoring types, symptoms, root causes, and triggering conditions. We obtained 12 findings and provided a series of valuable guidelines for future work on refactoring bug detection and debugging. Furthermore, our transferability study revealed 134 new bugs in the latest version of those refactoring engines. Among the 22 bugs we submitted, 11 bugs are confirmed by their developers, and seven of them have already been fixed.

CCS Concepts: • **Software and its engineering → Software reliability**.

Additional Key Words and Phrases: Refactoring Engine Bug, Refactoring, Empirical Study

ACM Reference Format:
Haibo Wang, Zhuolin Xu, Huaien Zhang, Nikolaos Tsantalis, and Shin Hwei Tan. 2025. Towards Understanding Refactoring Engine Bugs. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 55 pages. https://doi.org/10.1145/3747289

1 Introduction

Refactoring is defined as the process of changing a software system in such a way that it does not alter the external behavior of the software but improves its internal structure [4]. During software development, developers can perform refactoring manually, which is time-consuming and error-prone, or with the help of tools that automate activities related to the refactoring process [44]. Refactoring has been well studied as an efficient way to improve software quality [12, 32, 80] as well as an effective way to facilitate software maintenance and evolution [33, 77, 101]. Refactoring

*Corresponding Author

Authors' Contact Information: Haibo Wang, Concordia University, Montreal, Quebec, Canada, haibo.wang@mail.concordia.ca; Zhuolin Xu, Concordia University, Montreal, Quebec, Canada, zhuolin.xu@mail.concordia.ca; Huaien Zhang, Hong Kong Polytechnic University, Hong Kong, , China, cshezhang@comp.polyu.edu.hk; Nikolaos Tsantalis, Concordia University, Montreal, Quebec, Canada, nikolaos.tsantalis@concordia.ca; Shin Hwei Tan, Concordia University, Montreal, Canada, shinhwei.tan@concordia.ca.

ACM 1557-7392/2025/1-ART1
https://doi.org/10.1145/3747289

recommendation tools such as the built-in refactoring engines in IDEs (e.g., Eclipse [15], IntelliJ IDEA [27], and Netbeans [73]) have been widely used to facilitate software refactoring.

Despite the prevalence of these refactoring automation tools, developers often experience unexpected results due to bugs in the refactoring engine. Those bugs could silently change program behavior or induce inconsistencies in the code base, thus producing severe effects on real-world applications or improving the complexity of maintenance. Past research works [10, 21, 46, 64–68, 81] have tried to find bugs in these engines; however, effectively finding bugs in refactoring engines remains a challenge due to the diversity of refactoring types, complex refactoring implementations, and the lack of a general and in-depth understanding of bugs in the refactoring engines. Although these techniques have been shown to be able to detect some refactoring engine bugs, they still suffer from low effectiveness. Specifically, they spend significant resources and time in generating and executing test programs with a relatively low bug-revealing capability. In addition, there is still a lack of general and in-depth understanding of refactoring engine bugs. For example, we are unclear about how refactoring engine bugs are induced (i.e., root causes), how these bugs affect the software (i.e., bug symptoms), and how these bugs can be found (i.e., test oracles). To fill these gaps, this paper conducts the first systematic empirical study to analyze bugs in refactoring engines. In particular, we investigate the following research questions.

RQ1: (Refactoring Types) What kinds of refactoring are more likely to trigger refactoring engine bugs?
By identifying which refactoring tends to trigger bugs in refactoring engines, researchers could improve the robustness and reliability of refactoring engines by designing more effective automated testing tools focusing more on the bug-prone refactoring types. Refactoring engine developers could pay more attention to the implementation and validation of those bug-prone refactoring.

RQ2: (Bug Symptoms) What are the symptoms of these bugs? How do these bugs affect the refactoring and IDEs?
The symptoms could facilitate the understanding of the consequences of refactoring engine bugs, which help to triage them and assess their impacts. Moreover, these bug symptoms could help in designing refactoring engine testing techniques with effective test oracles.

RQ3: (Root Causes) What are the root causes of these bugs? What is the occurrence frequency of different root causes of refactoring engine bugs?
The root causes facilitate the understanding of the nature of refactoring engine bugs, which is helpful to detect, localize, and fix bugs. Moreover, it is worthwhile to explore the root causes specific to refactoring engine bugs and also investigate whether the conclusions on common root causes between refactoring engine bugs and traditional software bugs are consistent or not.

RQ4: (Triggering Conditions) What are the triggering conditions for the bugs in refactoring engines?
To guide and facilitate automated testing for refactoring engines, it is important to understand the triggering conditions. For example, understanding the characteristics of the input programs triggering the refactoring engine bugs could help researchers design more effective tools by generating more error-prone input programs.

RQ5: (Statistical Analysis) What is the correlation between each category pair?
Analyzing the relationship between different category pairs could further improve the understanding of refactoring engine bugs. For instance, by analyzing the relationship between input program characteristics and refactoring types, we can investigate which input program characteristics are more likely to result in certain refactoring failures. Thus, we could generate or prioritize the input program having certain characteristics for some refactoring to improve the

testing effectiveness and efficiency. To answer RQ5, we first run the Chi-squared test of independence [43] to see if the two categories are related. For example, the Chi-squared test for input program characteristics and refactoring types is 4374.049, and the p-value is 1.941e−6, which indicates that they are correlated (p-value < 0.05). Then, we perform a pairwise post hoc analysis to investigate which value pairs have statistically significant relationships. For example, the post hoc analysis shows that (Varargs[1], introduce parameter refactoring) are correlated with each other, indicating that performing introduce parameter refactoring for the method that contains Varargs parameter tends to be error-prone. Based on this finding, we could design a tool to generate the input programs or prioritize the input programs generated by existing tools that contain Varargs parameter when testing the introduce parameter refactoring, thus improving the effectiveness and efficiency. In addition, we also performed the same statistical analysis for the co-occurrence between (Root cause, Symptom), (Refactoring type, Root cause), (Refactoring type, Symptom), (Input program characteristic, Root cause), (Input program characteristic, Symptom), the detailed procedures and result are shown in Section 4.5.

Our study is based on the refactoring engines of three popular IDEs, namely Eclipse, IntelliJ IDEA, and Netbeans. We studied 518 bugs that were collected and labeled manually according to a systematic process. From our manual analysis of these refactoring engine bugs, we identified six root causes, nine bug symptoms, and eight main input program characteristics, including 38 subcategories, and obtained 12 main findings. Based on these findings, we provide a series of guidelines for refactoring engine bug detection and debugging.

In summary, this paper has made the following contributions:

- To the best of our knowledge, we conducted the first systematic study to investigate refactoring engine bugs. We study refactoring engine bugs from different perspectives (i.e., error-prone refactoring types, root causes, bug symptoms, and trigger conditions) and distill 12 findings. Table 1 concludes the key findings of our study. The findings of our study aim to encourage a better understanding of the refactoring engine bugs, which can potentially benefit the refactoring engine bug detection.
- By mining historical bug reports from the issue tracking systems of the refactoring engine, we curated a dataset including 518 bugs through a systematic labeling process, namely RefactorBench. The dataset contains categorized bug reports submitted by users of refactoring engines, which serve as the basis for our study and future research in this direction. To facilitate further studies in this area, we open source our data [79].
- We conducted a transferability study based on historical bug reports and our findings. As a result, we have found 134 new bugs in the latest version of the refactoring tools. Among the 22 bugs that we submitted, 11 bugs are confirmed by their developers, and seven of them have already been fixed.

2 Background

2.1 Refactoring Engine

Refactoring engines are integral components of integrated development environments (e.g., Eclipse, IntelliJ IDEA, and Netbeans), which provide semi-automated or automated support to help developers restructure and optimize their code. Although they differ in details and integration with each IDE, their overall workflows share similarities [19, 25, 56]. Figure 1 shows the general workflow of the refactoring engines. The input includes the input program and the configuration for a refactoring, and the output is the refactored program. The configuration in Figure 1 represents the configuration of the refactoring to be performed on an input program. For example, "Extract Local Variable" refactoring in Eclipse provides a user interface with options to allow developers to select whether to replace all occurrences of

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html

Table 1. Our key findings and implications.

Finding	Implication
Finding 1: "Extract" (149/28.76%), "Inline" (88/16.99%), and "Move" (80/15.44%) are the top three most error-prone refactoring types. Existing testing tools only cover about one-third (30/99) of the refactoring types.	Refactoring engine developers should focus on improving the reliability of the most error-prone refactoring, improving test coverage, and considering more diverse input programs. Current testing tools only cover about one-third of the refactoring types, leaving gaps that could be targeted for improvement.
Finding 2: "Compile Error" (242), "Crash" (106), and "Behavior Change" (66) are the top three most common symptoms of refactoring engine bugs.	Although some of the common symptoms (compile error, crash) can be easily detected via compiler or execution, there remains challenges in detecting bugs related to behavior change. Researchers and developers should enhance test oracles and focus on effectively identifying behavior changes.
Finding 3: Apart from the refactored code itself, other aspects such as warning messages and refactoring availability could also be error-prone. Current refactoring engines have not thoroughly taken into account various side effects that may be incurred by the refactoring.	Incorporating side effects, such as refactoring availability or warning messages, into bug detection strategies could provide a broader perspective on refactoring correctness. It is important to maintain the consistency between the refactored code and other software artifacts (e.g., code comments, and breakpoints).
Finding 4: Our study found that "Incorrect Transformations" (109) is the most common root cause for both and , accounting for 51 and 58 bugs, respectively.	Our study highlights the prominence of incorrect transformations as a root cause of bugs. Developers should work towards diverse input programs and leverage data-driven automation to minimize these issues. Researchers could investigate large language models (LLMs) to generate diverse input programs.
Finding 5: "Incorrect Flow Analysis" (97), and "Incorrect Type Resolving" (45) are also non-negligible root causes that are related to the ineffectiveness of the underlying analysis techniques.	The frequent issues of incorrect flow analysis and incorrect type resolving highlight the need for more robust analysis techniques in refactoring engines. Improving these areas could significantly reduce bugs caused by them.
Finding 6: "Incorrect Preconditions Checking" (62) is the third most common root cause. Preconditions need to be updated considering the introduction of new language features.	Preconditions need to be updated considering the introduction of new language features. Considering the evolution of programming language and automating the inference of preconditions for refactoring could be a promising research direction.
Finding 7 Most of the refactoring bugs (97.1%) could be triggered by the default initial input options of refactoring engines.	As most refactoring engine bugs could be triggered by the default configuration, researchers of testing techniques can focus on designing test programs instead of testing configuration-related refactoring bugs.
Finding 8a: Refactoring programs involving Java language features are more likely to trigger bugs in refactoring engines, they take up 64.4% of our studied bug reports.	More testing efforts should be paid to the input programs with language-specific features, considering 64.4% of our studied bug reports are related to it, and its prevalence in the OSS repositories. Language features, particularly lambda expressions, generics, and Enum contribute to most refactoring bugs.
Finding 8b: Lambda expression (39/14.4%), Java generics (38/14.1%), and Enum (27/10.0%) are the top three language features triggering refactoring engine bugs due to the complexity of type inference, limited flow analysis, and complicated usage scenario.	
Finding 9: Input programs having complex class relationships are more likely to result in refactoring engine failure. Among these, refactoring involving inner class (19/7.0%) and anonymous class (15/5.6%) are the top two most bug-prone.	Complex class structures, such as inner and anonymous classes, are prone to triggering bugs. When designing the templates or rules to generate input programs, or mining input programs for testing, researchers should focus on these language features.
Finding 10: Annotation-induced (29/10.7%) refactoring engine bugs are the third most in our studied bug reports with input program. The reasons include: (1) Lack of consideration of the semantics of annotations, (2) Complexity of annotation processing, which may be processed at various phrases: compile-time (e.g., @Override), runtime (e.g., @Deprecated), and deployment (e.g. @WebServlet in Java Servlet API), (3) Complex transformations for annotations.	Annotations add complexity due to their semantic meaning and wide-ranging effects. Refactoring engines should be enhanced to handle these constructs more effectively, particularly by improving transformations that respect class and annotation intricacies. Future research could explore how refactoring engines can be improved to handle them better.
Finding 11: Refactoring involving comments interleaved with code might cause refactoring to fail (18/6.7% of our studied bugs). The reasons include: (1) Comment-code dependency, and (2) Semantic gaps between code and comments, indicating the needs for comment-aware auto-refactoring tools.	There is a gap in research regarding comment-aware refactoring. Researchers should investigate new methodologies for creating tools that can bridge the semantic gap between code and comments, enabling more robust refactoring that considers code-comment dependencies.
Finding 12: Through our statistical analysis, we conclude that (Root Cause, Symptom), (Input program characteristic, Refactoring type), (Refactoring type, Root cause), (Input program characteristic, Root cause) and (Input program characteristic, Symptom) are correlated, while (Refactoring type, Symptom) is not correlated.	By studying the correlation between each category pair, we can further improve the understanding of refactoring engine bugs. Findings could be leveraged to design more effective testing tool.

the selected expression with references to the newly extracted local variable. We explain each step in the refactoring engine as follows:

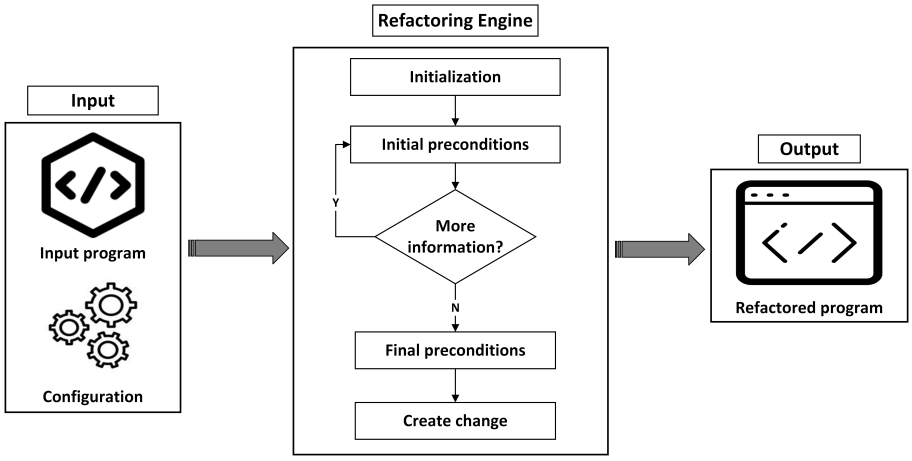

Fig. 1. The general workflow of refactoring engine.

1. Initialization. In this stage, the refactoring engine initializes the refactoring operation, sets up the necessary context, and identifies the elements to be refactored.

2. Initial condition checking. This phase is responsible for checking whether the prerequisites to start the refactoring are met and checking conditions such as the validity of the selection. If the initial condition check fails, the refactoring is prevented from proceeding further and warnings are displayed to the users. For example, in the initial condition checking stage of Make Static refactoring, the method to be refactored should not be a Constructor method[2].

3. Additional information. In this phase, the refactoring engine collects any additional user inputs or configurations (e.g., user can choose whether "Replace all occurrences of the selected expression with references to the local variable" for the "Extract Local Variable" refactoring) required for the refactoring. For some refactoring, additional information might be required to perform the refactoring (e.g., the new method name when performing "Extract Method" refactoring).

4. Final condition checking. After the initial condition check has been met and all necessary information is collected to perform the refactoring, the final condition checking can start to check the remaining preconditions. Examines the potential changes resulting from refactoring and verifies that refactoring can be carried out successfully without bad effects (e.g., potential conflicts with existing code, behavior-preserving or quality preconditions [75]). If checking fails, the users will get a warning or error message. In the final condition checking stage, the refactoring engine will modify the input program depending on the refactoring to be applied, then it checks if these changes will bring some bad effects like name conflicts. If all the final preconditions are fulfilled, the refactoring engine will perform the create change step to the input program to produce a refactored program; otherwise, some warning messages will be shown to users. For the initial preconditions check, it only checks if the input program fulfills some prerequisites for the refactoring to be applied. It will not modify the input program to check if the refactored program can fulfill some conditions. The final condition checking phase focuses on verification and error detection to ensure the safety and feasibility of the refactoring operation. For example, Make Static refactoring cannot be executed if the refactored method is overridden in a child class because a static method cannot be overridden[2].

[2]https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/590

5. Create change. This phase is responsible for generating the actual code modifications. The refactoring engine needs to compute the specific changes needed to apply the refactoring, create some change objects representing the modifications to be applied, and transform the code. For example, when performing Make Static refactoring for the program, the refactoring engine needs to (1) modify the method declaration (2) update method invocations and (3) handle related changes (e.g. update the Java documentation for the refactored method).

Steps 3 to 5 can be executed repeatedly (e.g., when the developer goes back from the preview page).

The types of refactoring supported in IntelliJ IDEA (2024.1.2), Apache NetBeans IDE 22, and Eclipse (2024-03) are shown in Figure 2, Figure 3, and Figure 4, respectively. Detailed information for each refactoring is available in the official user manual documents of different IDEs[3][4][5]. As we can see in Figure 5, most supported refactoring is common in these IDEs, including frequently used refactoring such as rename, extract, inline, pull-up, etc. However, there are few refactorings that are unique to specific IDEs. For example, "Make Static" refactoring converts an inner class or an instance method to a static one[6], while it is supported in Eclipse (2024-03) and IntelliJ IDEA (2024.1.2), it is not available in Netbeans (Apache NetBeans IDE 22).

2.2 Motivation Example

In this chapter, we introduce a bug related to Make Static refactoring as an illustrative example to explain (1) why existing refactoring engine testing tools fail to trigger the bug and (2) the insights we could get by analyzing existing bug reports.

Eclipse Bug 1083[7]. The output of the code before refactoring in Listing 1 of Figure 6 is "Counter: 15". However, after performing Make Static refactoring (RQ1 refactoring type) using the refactoring engine of Eclipse for the method "toBeRefactored()", the output of the refactored program in Listing 2 becomes "Counter: 5". The refactored program does not contain any syntax errors and Eclipse just silently changes the program behavior without any warning (RQ2 symptom). This happens because Eclipse erroneously interprets a method call within an anonymous class inside the refactored method as a call on "this" and thus changes the method invocation to be performed on the added input parameter of the method (RQ3 root cause). However, the call to the method "toCall()" was made on an instance of the anonymous inner class rather than the outer instance passed to the "toBeRefactored()" method. This bug could potentially have multiple effects, as the developer stated — "This is (1) semantically incorrect and (2) leads to compile errors if the type of added parameter does not provide the method"[7].

The SAFEREFACTOR-based tools [46, 64–68] cannot identify this bug because they can only generate test cases for the common methods before and after the refactoring in the project, which means that they cannot exercise the changed methods (e.g., refactored methods). ASTGen [10] fails to detect this bug because it does not have predefined templates for Make Static refactoring. Based on the analysis of the current Eclipse bug report, we conducted a transferability study (refer to Section 5 for details) by cross-validating this bug in both IntelliJ IDEA and Netbeans, and successfully discovered a new bug in the latest version (2024.1.2 at the time of our study) of IntelliJ IDEA. We have submitted a bug report[8] to the IntelliJ IDEA issue tracker system, it has been confirmed and fixed by the engine developers.

[3] https://www.jetbrains.com/help/idea/refactoring-source-code.html
[4] https://netbeans.apache.org/tutorial/main/kb/docs/java/editor-inspect-transform/
[5] https://help.eclipse.org/2024-03/index.jsp
[6] https://www.jetbrains.com/help/idea/make-method-static.html
[7] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1083
[8] https://youtrack.jetbrains.com/issue/IDEA-354116

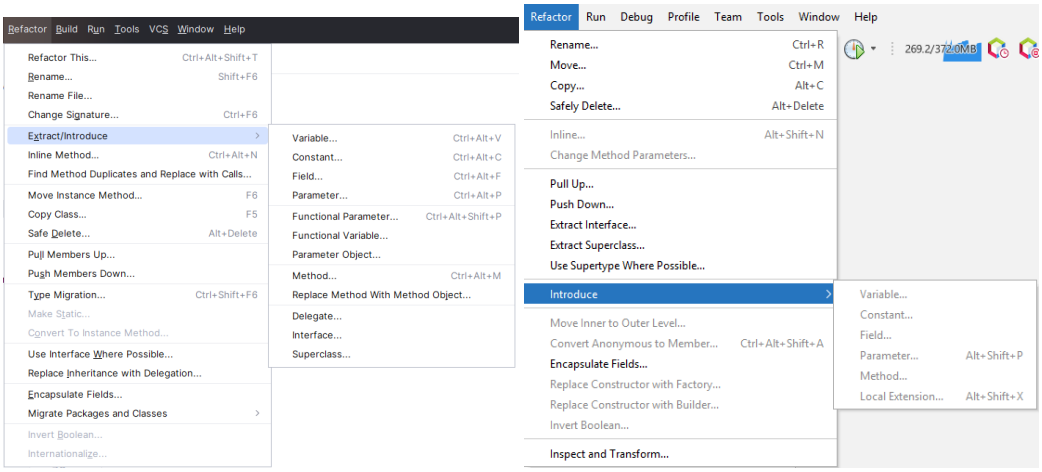

Fig. 2. Refactorings supported by IntelliJ IDEA (2024.1.2).

Fig. 3. Refactorings supported by Apache NetBeans IDE 22.

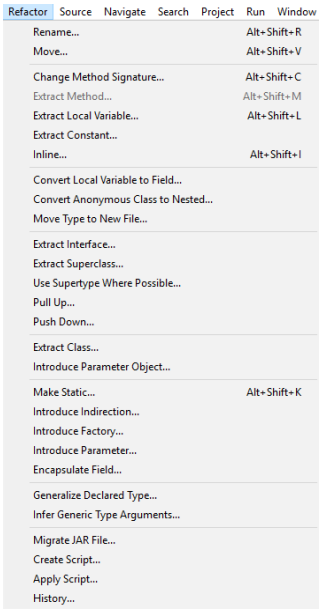

Fig. 4. Refactorings supported by Eclipse (2024-03).

Fig. 5. Refactoring types that are supported in the latest version of different IDEs at the time of our study.

In addition to the bug in Figure 6, there are still some other bugs[9][10][11] caused by Make Static refactoring. Those bugs could silently change the program behavior, resulting in compile errors or inducing inconsistencies in the code base, thus producing unexpected behavior in real-world applications or making it difficult to maintain. Analyzing those

[9] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1043
[10] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1044
[11] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1045

```
1  public class Foo {
2      private int counter;
3      public Foo(int initialCounter) {
4          this.counter = initialCounter;
5      }
6  -   void toBeRefactored() {
7  -       new Foo(counter + 10) {
8              void toImplement() {
9  -               toCall();
10             }
11         }.toImplement();
12     }
13     void toCall() {
14         System.out.println("Counter: " + counter);
15     }
16     public static void main(String[] args) {
17         Foo foo = new Foo(5);
18 -       foo.toBeRefactored();
19     }
20 }
```

Listing 1. Code before make static refactoring.

```
1  public class Foo {
2      private int counter;
3      public Foo(int initialCounter) {
4          this.counter = initialCounter;
5      }
6  +   static void toBeRefactored(final Foo foo) {
7  +       new Foo(foo.counter + 10) {
8              void toImplement() {
9  +               foo.toCall();
10             }
11         }.toImplement();
12     }
13     void toCall() {
14         System.out.println("Counter: " + counter);
15     }
16     public static void main(String[] args) {
17         Foo foo = new Foo(5);
18 +       Foo.toBeRefactored(foo);
19     }
20 }
```

Listing 2. Code after make static refactoring.

Fig. 6. Eclipse-1083: Make Static refactoring erroneously changes method call.

bug reports could provide us with a deep understanding of the refactoring engine bug triggering conditions, testing oracle design, and input program generation. This paper makes the first attempt to comprehensively study refactoring engine bugs, paves the way for related future research, and provides guidelines for engine developers and researchers to test and improve the reliability of refactoring engines.

3 Methodology

3.1 Dataset Collection

In this study, we use the three most popular refactoring engines as subjects, including (1) Java Development Tools (JDT) from Eclipse, (2) the Java refactoring component of IntelliJ IDEA, and (3) Netbeans refactoring engine. As Java language support for Visual Studio Code (VSCode) is built upon Eclipse JDT [45], we choose to focus on Eclipse and exclude Visual Studio Code (VSCode) as our study object. There are some categories of bug reports with different purposes, such as feature requests and questions. Hence, we need to identify the bug reports that are refactoring engine bugs only. Specifically, following existing studies [59, 70], we collect bug reports with titles or discussions that contain at least one refactoring engine bug-relevant keyword (i.e., "refactoring" and "refactor") before the time we conduct our study (1 July 2024). The keywords searching results in 3273 bug reports from Eclipse, 4965 bug reports from IntelliJ IDEA and 572 bug reports from Netbeans. Our study aims to investigate the characteristics of refactoring engine bugs, so we focus on bugs that are fixed and not duplicated, together with their corresponding patches [59, 70, 110]. Specifically, we consider a bug is fixed if its "Resolution" field is set to "FIXED" and the "Status" field is set to "RESOLVED", "VERIFIED" or "CLOSED" in the bug repositories of Eclipse and Netbeans in Bugzilla. We leverage the advanced search user interface in Bugzilla[12] to retrieve historical bug reports that fulfill our criteria. Then, we build a web crawler to parse and download the searched results. For issues after a specific date (Jan 2022 for Netbeans, and Apr 2022 for Eclipse JDT), both Eclipse JDT and Netbeans started to migrate their issue trackers to GitHub. To get a complete list of bug reports, we also crawled issues and patches from their GitHub repositories using the GitHub APIs [20]. For each repository, we search for the closed issues (resolved bug reports) with the same keywords, since these issues have been confirmed by the developers and are more likely to contain fixes. We further

[12] https://bugs.eclipse.org/bugs/query.cgi

Table 2. Statistical information of our dataset.

Tool	Duration	#CB	#RB	#IP	#IPP
Eclipse	2005-03-16 ~ 2024-06-12	841	289	240	122
IntelliJ IDEA	2016-03-23 ~ 2024-03-22	601	268	209	204
Netbeans	2012-04-08 ~ 2024-04-22	209	80	69	0
Total	-	1651	637	518	326

CB : Collected bug reports; **RB** : Refactoring engine bug reports; **IP** : Refactoring engine bug reports containing input program; **IPP** : Further filter refactoring engine bug reports containing both input program and patch. Note that we derive our RQ1, RQ2, and RQ5 on the IP, RQ3 and RQ4 on the IPP.

searched for issues that contain at least one corresponding commit. The goal is to get a list of related and closed issues, which have been fixed by the developers. To remove the bug reports that exist in both GitHub and Bugzilla, we de-duplicate them based on their bug ID and title. We only collect fixed bug reports because (1) they are considered to be more important by developers, and (2) such reports contain more information (e.g., patches, and discussions among developers) about the involved bugs, which allows us to have a better understanding of these bugs. For IntelliJ IDEA, we collected fixed bugs from its issue tracker through its issue searching user interface. Then we crawled the searched results from webpages using Selenium framework[13]. Since the patches are not included in the bug reports in the IntelliJ IDEA issue tracker [29], we need to identify the revisions that correspond to these fixed bugs. As IntelliJ IDEA developers usually add the bug report ID as a marker in the commit message, we use the GitHub APIs [20] to search for commits where the commit message contains the bug report ID in the JetBrains/intellij-community [28] repository. The refactoring engine developers usually marked a bug report as duplicated if it has been reported before. During our dataset collection stage, we filter out the duplicated issues if they are marked as "Duplicated".

Table 2 presents detailed information about our dataset. In total, we collect 1651 initial bug reports that meet our search criteria before the time we conduct our study (1 July 2024). The column "Duration" in Table 2 indicates the earliest and latest dates for these bug reports. The dates denote the duration of bug reports that satisfy our selection criteria. The "#CB" column presents the detailed number of bug reports for each refactoring engine. Then, two authors manually analyzed these reports (refer to Section 3.2 for the detailed process). Among them, we identified 637 bug reports related to refactoring engine bugs. In which, 518 bug reports contain input programs. These bug reports form the basis for our RefactorBench dataset. Due to the migration of the bug tracking systems, some patch links are out of date and no longer available. Finally, we collected 326 bug reports that contain both the input program and the patch (IPP dataset). Compared to Eclipse and IntelliJ IDEA, Netbeans tends to contain fewer refactoring engine bug reports mainly because it is not as widely used as the other two [26]. We derived our RQ1 (Refactoring Types), RQ2 (Bug Symptoms), and RQ4 (Trigger Conditions) based on the RefactorBench dataset, since the input programs in those bug reports are confirmed by the engine developers that they can trigger the bugs, which means they are valuable for our research. As for RQ3 (Root Causes) and RQ5 (Statistical Analysis), they require not only analyzing the input programs and discussions in the bug reports, but also analyzing the patches to determine the root causes; thus we use the IPP dataset.

3.2 Classification and Labeling Process

We investigate each bug from four aspects: 1) the refactoring type of the bug, 2) the root cause of the bug, 3) the symptom that the bug exhibits, and 4) the characteristics of the input program triggering the bug. To build the taxonomies, we

[13] https://en.wikipedia.org/wiki/Selenium_(software)

adopted the open coding approach [31]. Specifically, the preceding analysis was done by an iterative process. In each iteration, 10% bug reports were selected, and the two annotators independently studied each of them. According to their own understanding, they independently labeled each bug with the categories of refactoring types, symptoms, root causes, and input program characteristics. Then, these two annotators cross-validated and discussed the labels until they reached consensus on the categorized results. When they could not reach consensus, a third annotator participated in the discussion to help make the final decision. Such an iteration was repeated ten times (i.e., labeling 10% ~ 100% of bug reports with an interval of 10%) until all the bugs were analyzed. This manual analysis process requires considerable domain-specific knowledge of both Java programming language and refactoring engine implementation, which took us around six person months. Following existing approaches [18, 24, 59, 72, 82, 85, 86, 103, 105, 107], we measured the inter-rater agreement among the annotators via Cohen's Kappa coefficient. Particularly, the Cohen's Kappa coefficient was nearly 40% for the first 10% of labeling results, and thus we conducted a training session about labeling. After that, two authors labeled 20% of bug reports (including the previous 10%), and Cohen's Kappa coefficient was computed as 85%. After further discussion of the disagreements, Cohen's Kappa coefficient was always more than 90% in subsequent labeling attempts. Finally, all bugs were labeled consistently. During the labeling process, we filtered out the irrelevant bug reports (e.g., feature requests). We derived the taxonomies of symptoms, refactoring types, and input program characteristics using the same procedures as we described above.

For the taxonomy of root causes, we first referred to the categories from existing research on bug understanding and detection [11, 48, 59, 99], however, we only managed to adopt two categories: "Overly Weak Preconditions" and "Overly Strong Preconditions" in our study. Since the other initial root causes are from other domains like deep compiler bugs and bugs in Android Apps instead of refactoring engines, we cannot use them directly. Hence, for the remaining root causes listed in Section 4.3, we derived them from our own study, representing the unique characteristics of refactoring engine bugs. As for the symptoms, we adopt behavior-changing symptoms from SAFEREFACTOR, and compile error, crash, and incorrect warning messages symptoms from ASTGen. The remaining symptoms described in Section 4.2 are obtained through our analysis. For the input program characteristics, there is no existing work to analyze the input program characteristics for refactoring engines, so we developed the taxonomy of input program characteristics by ourselves. The refactoring types are extracted from the bug reports, and we categorized them according to the existing refactoring menus in refactoring engine. Notably, we are the first to study the error-proneness of the refactoring types in refactoring engines.

To answer RQ1 error-prone refactoring type, we first merged and categorized concrete refactoring types into higher-level categories according to existing refactoring menus in Eclipse (2024-03) because it supports the greatest number of refactoring types, then we calculated the number of bug reports in each category. After that, we analyzed the refactoring types that are covered by existing testing approaches. For RQ2 bug symptom, RQ3 root cause, and RQ4 triggering condition, we first clustered bug reports according to their labels, then we calculated the number of bug reports in each cluster and conducted quantitative analysis. For each category, we performed qualitative analysis to gain a comprehensive understanding. To perform statistical analysis for RQ5, we first performed a Chi-squared test of independence, then a pairwise post hoc analysis to investigate which value pairs have statistically significant relationships.

3.3 Information Extraction and Labeling using LLM

To improve the scalability of the labeling process, we explored the possibility to leverage the power of LLM together with few-shot learning to automate the labeling process. This LLM-assisted approach is inspired by an existing study that

Table 3. The prompt template used to label historical bug reports.

You are a software testing expert. I will give you some historical bug reports for the refactoring engines. You need to extract the following information from the bug reports: 1. Refactoring type; 2. Bug symptom; 3. Input program characteristic. The following are examples: **{Example}.** The extracted information format should be: **{Format}**.

used LLM to filter irrelevant forum posts [23]. Instead of using LLM to filter out irrelevant posts as in prior study [23], we use LLM to extract the necessary information (e.g., refactoring type) for subsequent analysis. Table 3 shows the prompt used to label historical bug reports. Specifically, we asked LLM to extract the refactoring type, bug symptom, and input program characteristics from the historical bug reports. For the root cause analysis, we still adopt the manual method, since labeling it involves understanding and analyzing the source code of the bug-fixing patch, which is usually complex and difficult to automate. We adopt ten-shot learning for this step by manually constructing ten examples and then incorporating the examples into the prompt template to instruct LLM to extract critical information for the rest of bug reports. We used OpenAI API[14] to programmatically invoke ChatGPT. The underlined model is GPT-4o-mini with the default settings[15].

To evaluate the effectiveness of LLM-based labeling, we randomly sampled 100 manually labeled historical bug reports from our dataset. Then, we inspected the automated labeling results of those 100 historical bug reports against the manually constructed ground truth. We also evaluated the time cost for automated labeling. The time was calculated by the end-to-end duration — from sending the request to receiving LLM-labeled results. For refactoring type, the LLM-based labeling achieved a high precision, in total, 98 out of 100 bug reports are correctly extracted. For the bug symptom, 80 out of 100 bug reports are identical to the ground truth. We manually checked the differences to identify the reasons behind the inaccuracy. Our manual analysis revealed that the inaccuracy is mainly due to some symptoms not being explicitly mentioned in some bug reports, causing the failure of LLM in effectively identifying the symptoms. For example, IDEA-104536[16] mentions that there exists red code after inline variable. During our manual labeling, we can observe that the red code happens in IntelliJ IDEA because the refactored program contains a syntax error, but the extracted symptom by LLM is "code changes", which is incorrect. For the input program characteristic, 60 out of 100 bug reports are the same as manually labeled. After analyzing the failed ones, we found out that LLM extracted refactoring-irrelevant input program characteristics in the input programs (i.e., these characteristics are in the input programs, but they appear only in the code blocks which are not involved in refactoring). When we manually extracted the input program characteristics, we would first localize the refactoring-relevant program elements before analyzing their characteristics. In future, we think that the results could be further improved by combining LLM together with program analysis techniques (e.g., program slicing), but these techniques usually require the input programs to fulfill some prerequisites (e.g., compilable), which is beyond the main focus of our study. The total time cost for automated labeling the 100 historical bug reports is 225 seconds, leading to 2.25 seconds average time for each bug report. Overall, LLM-based labeling shows promising results for certain research questions, but a more thorough analysis is required

[14] https://platform.openai.com/docs/overview
[15] https://platform.openai.com/docs/api-reference/chat/create
[16] https://youtrack.jetbrains.com/issue/IDEA-104536

for some research questions (e.g., root causes). Hence, to conduct a thorough study of refactoring engine bugs, we conduct all subsequent analyses based on the manually labeled dataset.

4 RESULTS AND ANALYSIS

4.1 RQ1: Refactoring Types

Table 4 shows the statistics of the bugs found in different refactoring types. To have a clearer and more organized view of various refactoring, we merge and categorize these concrete refactoring types into higher-level categories. Specifically, the "Refactoring" column gives the concrete refactoring types with the corresponding total number of bug reports. All these refactoring types are supported by our studied refactoring engines. We further merge some common refactoring types into "Subcategory" based on their operations (e.g., extract, inline, move, etc.) and the granularity of the program to be refactored (e.g., class, method, and variable). Finally, refactorings with the same operations are grouped into the "Category" column, which gives the highest-level refactoring category along with the number and percentage of bug reports.

From a high-level perspective, as shown in the "Category" column in Table 4, "Extract" (149/28.76%), "Inline" (88/16.99%), and "Move" (80/15.44%) are the top three most error-prone refactoring transformations, among which the number of bug reports for the "Extract" refactoring nearly double the number of "Inline" and "Move" refactoring. During manual analysis, we deduce two reasons for the types of refactoring that are prone to errors: (1) popularity of refactoring (our result is in line with previous studies [1, 22, 49, 55] that revealed that refactoring of extract, inline, and move is among the operations most commonly applied). (2) those refactoring types are more general and serve various purposes (e.g., facilitate reusability, improve readability, remove duplication, etc.) [1, 55, 61], thus they can be performed in some more complicated usage scenarios (e.g., extract a static method from a lambda expression inside an inner class [93]), which makes it more challenging for the refactoring engines to take into account all the complicated usage scenarios. For the concrete refactoring types supported by our studied refactoring engines in the "Refactoring" column, "Inline Method" (46), "Move Method" (42), and "Extract Method" (40) trigger the top three number of bugs, following by "Change Method Signature" (38), and "Extract Variable" (37). Developers are more familiar with these types of refactoring and are more likely to automate them with the support of IDEs according to a previous study [22] on the refactoring usage conducted by JetBrains IDE development teams. Although most bug reports in our study only involve one refactoring, there exist five bug reports mention that the same bug appeared in more than one refactoring type (e.g., "Extract variable" and "Extract Constant" do not handle "Text Block" in Eclipse 4.12 [96]) so we label them as "multiple" and categorize them into row "Others" in the "Refactoring" column. For some refactoring types (e.g., "Infer Type Arguments" and "Make Static") which cannot be categorized into a more general category, we put them in the "Others" category. Among the refactoring types in "Others", if the number of bug reports is smaller than five (e.g., "Surround With Try/Catch"), we merge them into the last row named "Others" in the "Refactoring" column.

Refactoring types are covered by existing testing approaches. We further analyze whether existing approaches [10, 21, 46, 64–68] for testing refactoring engines can be used to automatically identify bugs for each refactoring type. Our analysis revealed that the existing approaches only cover part of the refactoring types in Table 4. ASTGen [10] focuses on testing only eight refactoring types (marked with a superscript "A" in Table 4 "Refactoring" column) and the top three most error-prone refactoring types have not been included in supported refactoring types. This is because ASTGen relies on manually designed program templates for each type of refactoring, which requires domain knowledge and incorporates various characteristics of input programs (we study and answer this question in Section 4.4) to effectively

Table 4. Statistics of bugs found across different refactoring types.

Category	(#)	(%)	Subcategory	(#)	Refactoring	(#)
Extract	149	28.76%	Inheritance Refactoring	61	Pull Up[A,S,G]	31
					Push Down[A,S,G]	12
					Extract Interface[G]	8
					Extract Class[S,G]	5
					Extract Superclass[G]	5
			Extract Variable	47	Extract Variable[G]	37
					Extract Constant[G]	10
			Extract Method	41	Extract Method[S,G]	40
					Extract Delegate	1
Inline	88	16.99%	Inline Method	46	Inline Method[G]	46
			Inline Constant And Variable	28	Inline Variable[G]	25
					Inline Field	2
					Inline Constant[G]	1
			Inline Class	13	Inline Class	11
					Inline Interface	2
			Inline Others	1	Inline Expression	1
Move	80	15.44%	Move Method	43	Move Method[S]	42
					Move Instance Method[G]	1
			Move Class	30	Move Type To New File[G]	20
					Move Inner Class To Outer Level[A]	6
					Move Class	4
			Move Constant And Variable	7	Move Field	4
					Move Constant	2
					Move Parameter	1
Rename	49	9.46%	Rename Method	20	Rename Method[A,S,G]	20
			Rename Constant And Variable	18	Rename Field[A,S,G]	9
					Rename Variable[S,G]	9
			Rename Class	10	Rename Class[A,S]	9
					Rename Interface	1
			Rename Other	1	Rename Enum	1
Change Signature	40	7.72%	Change Method Signature	38	Change Method Signature[A,S,G]	38
			Change Class Signature	2	Change Class Signature	2
Introduce	25	4.83%	Introduce Variable	15	Introduce Variable	9
					Introduce Field	4
					Introduce Constant	2
			Introduce Method	9	Introduce Factory[G]	5
					Introduce Indirection[G]	2
					Introduce Local Extension	1
					Introduce Method	1
			Introduce Class	1	Introduce Parameter Object[G]	1
Convert	24	4.63%	Convert Constant And Variable	10	Convert Variable	4
					Convert String To Textblock	4
					Convert Boolean	1
					Convert To Automic	1
			Convert Class	8	Convert Anonymous To Inner ClassClass[G]	7
					Convert Class To Record	1
			Convert Method	6	Convert To Switch Expression Expression	2
					Convert To Enhanced For Loops	2
					Convert Anonymous To Lambda	1
					Convert To Instance Method	1
Replace	10	1.93%	Replace Class	6	Replace Constructor With Builder	2
					Replace Constructor With Factory	2
					Replace Constructor With Factory Method	2
			Replace Method	4	Replace With Lambda	2
					Replace Anonymous With Lambda	1
					Replace Foreach With For	1
Other	53	10.23%	Others	53	Encapsulate Field	7
					Infer Type Arguments[G]	7
					Make Static	5
					Use Supertype Wherever Possible[G]	5
					Others[A,S,G]	29

The superscript 'A' indicates the refactoring type is covered by ASTGen; 'S' indicates that current refactoring is covered by SAFEREFACTOR-based tools; 'G' means current refactoring is covered by [21]. Note that some covered refactoring types are merged into the "Others" row in the "Refactoring" column, so if there is one refactoring inside is covered, we mark the "Others' with the superscripts.

```
1  public class TestCase{
2      public void main() {
3          class T {
4              public T() {}
5          }
6  -        foo();// inline method foo()
7      }
8      public void foo() {
9          class T {
10             T t;
11             public T() {}
12         }
13     }
14 }
```

Listing 3. Code before refactoring.

```
1  public class TestCase{
2      public void main() {
3          class T {
4              public T() {}
5          }
6  +        class T {// syntax error
7  +            T t;
8  +            public T() {}
9  +        }
10     }
11     public void foo() {
12         class T {
13             T t;
14             public T() {}
15         }
16     }
17 }
```

Listing 4. Code after refactoring.

Fig. 7. IDEA-332489: inline method refactoring leads to naming conflicts in IntelliJ IDEA 2023.3.4.

trigger the bugs of the refactoring engine. Approaches based on SAFEREFACTOR [46, 64–68] also cover a limited number of refactoring types (marked with a superscript 'S'). Meanwhile, Gligoric et al. [21] proposed an approach that supports the testing of 23 refactoring types (marked with a superscript 'G') available in the Eclipse 4.2 refactoring menu; however, they are still limited in (1) Some applicable elements for each refactoring are not covered. For example, they test instance method and static method for the move refactoring but other elements such as class, constant, and field that are currently out of their scope; (2) Refactoring types that are unique in IntelliJ IDEA and Netbeans (e.g., "Convert To Enhanced For Loops") are not covered because they only test Eclipse; (3) Refactoring types added in the newer version (after version 4.2) of Eclipse are not covered (e.g., "Convert String to Text Block"). In total, 30 of the 99 refactoring types in Table 4 are covered by at least one of the existing refactoring testing tools. As the remaining refactoring types have not been covered, developers of refactoring engines for these unsupported types have to resort to testing them manually. This indicates a gap in existing testing tools and the need to design a general testing approach that can support more refactoring types.

Finding 1: "Extract" (149/28.76%), "Inline" (88/16.99%), and "Move" (80/15.44%) are the top three most error-prone refactoring types. Existing testing tools only cover about one-third (30/99) of the refactoring types.

4.2 RQ2: Bug Symptom

Based on the classification and labeling process described in Section 3.2, we identify several symptoms of refactoring engine bugs below:

Compile Error (242). The refactored program is syntactically incorrect, resulting in compile errors. Figure 7 shows that in IntelliJ IDEA 2023.3.4, if the refactoring of the inline method is applied to the method "foo()" in line 6 of the input program in Listing 3, the refactored program will not be compilable due to syntax error. As shown in Listing 4, the refactored program contains two inner classes both named "T" in the "main()" method, and this name conflict would result in a syntax error at line 6[17].

Crash (106). The refactoring engine terminates unexpectedly during refactoring and throws an exception. For example[18], for the input program in Listing 5, performing the inline method refactoring for the method "format()" at line 8 would

[17] https://youtrack.jetbrains.com/issue/IDEA-332489
[18] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/197

```
1  public class Main {
2      public static String format (String... input)
           {
3          return "";
4      }
5      public static void main(String[] args) {
6          int value = 0;
7          String message = switch (value) {
8              case 0 -> format("");// inline format
                   ()
9              default -> "";
10         };
11     }
12 }
```

Listing 5. Input program.

```
1  java.lang.reflect.InvocationTargetException
2          at org.eclipse.jface.operation.
                 ModalContext.run(ModalContext.java
                 :395)
3          ...
4  Caused by: java.lang.NullPointerException: Cannot
       invoke "org.eclipse.jdt.core.dom.rewrite.
       ListRewrite.insertAt(org.eclipse.jdt.core.dom
       .ASTNode, int, org.eclipse.text.edits.
       TextEditGroup)" because "this.fListRewrite"
       is null
5          at org.eclipse.jdt.internal.corext.
                 refactoring.code.CallInliner.
                 addNewLocals(CallInliner.java:569)
6          ....
```

Listing 6. Stacktrace in the error log.

Fig. 8. Eclipse-197: NPE when using inline refactoring on method with varargs in switch expression.

result in a crash for Eclipse before version 4.25. The stack trace in the corresponding error log is shown in Listing 6 (i.e., a "NullPointerException" is thrown). We consider a crash as a bug symptom because: (1) the crash occurs because of defects in the refactoring engine. For example, after studying the patch[19] for the bug in Figure 8, we found that this bug occurs because when a refactoring engine performs program transformation by rewriting the AST of the input program, inlining method within the switch expression is not fully supported. (2) The developers of the refactoring engines have confirmed and fixed this kind of bug report.

Behavior Change (66). The refactoring engine successfully applies the corresponding refactoring and produces a refactored program without any syntax error. However, the refactored program is not behavior-preserving, which violates the definition of refactoring [95]. For example[20], in Eclipse before version 4.28, for the input program in Listing 7 of Figure 9, 1) selecting the expression "Lines[i]" at line 7, then 2) performing "extract local variable" refactoring, and 3) choosing to replace all occurrences of the selected expression with references to the newly extracted local variable through the user interface would result in the refactored program in Listing 8. The refactoring is incorrect because the original values of "Lines[i]" at line 7 and line 8 of the Listing 7 are different from "Lines[i]" at line 13 because of the increment of "i" in method "inc()". Consequently, replacing "Lines[i]" at line 14 in Listing 8 with extracted variable "x" is incorrect. The expression replaced on line 14 of Listing 8 depends on the variable "i" that could be changed by statements (i.e., "inc()" on line 12) between the initialization of the variable and the references to the variable, thus the behavior is different for the program before and after the extract local variable refactoring.

Incorrect Warning Message (24). The refactoring engine gives an incorrect or confusing warning message to prevent the refactoring from being performed. For example, applying "Change Method Signature" refactoring in Eclipse 4.17 for the method "foo()" inside a *Record* [50] type declaration on line 2 in Listing 9 of Figure 10 would raise an incorrect warning message — "could not resolve type int". As the developer complained "continue renames correctly, but the error is unwarranted"[21].

Failed Refactoring (23). The refactoring engine either makes no change to the original program or only partially completes the refactoring. In this case, the refactored program is still compilable and behavior-preserving, but the result does not align with the user's intention since the refactoring either changes nothing at all or has only been partially completed. Figure 11 shows an example[22] where the extract local variable refactoring is partially completed in Eclipse

[19] https://github.com/HannesWell/eclipse.jdt.ui/commit/8f198d83ed5211235084b67ffa0104e64fe215bb
[20] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/432
[21] https://bugs.eclipse.org/bugs/show_bug.cgi?id=566866
[22] https://bugs.eclipse.org/bugs/show_bug.cgi?id=104293

```
1  String[] Lines;
2  int i;
3  private void inc() {
4      i++;
5  }
6  ...
7  -log.warn(Lines[i]);// extract local variable
8  ...
9  -log.warn(Lines[i]);
10 ...
11 inc();
12 ...
13 -log.warn(Lines[i]);
```

Listing 7. Code before refactoring.

```
1  String[] Lines;
2  int i;
3  private void inc() {
4      i++;
5  }
6  ...
7  +String x= Lines[i];
8  +log.warn(x);
9  ...
10 +log.warn(x);
11 ...
12 inc();
13 ...
14 +log.warn(x);// behavior change
```

Listing 8. Code after refactoring.

Fig. 9. Eclipse-432: improve the safety of "Extract Local Variable" refactoring by identifying statements that may change the value of the extracted expressions.

```
1  record R() {
2          public int foo() {// change foo to bar
3                  return 0;
4          }
5  }
```

Listing 9. Input program.

Fig. 10. Eclipse-566866: incorrect warning message when perform change method signature refactoring in Eclipse 4.17.

3.1. As shown in the Listing 10, there are two identical expressions (i.e., asList(side)) in lines 6 and 9 of the input program. When a user performs "Extract Local Variable" on one expression and selects the checkbox "Replace All Occurrences", Eclipse only replaces the selected expression with the extracted variable, but the other identical expression (in line 11 of Listing 11) remains unchanged. As shown in the Listing 11, the expression in line 8 is replaced with the extracted variable; however, the same expression in line 11 is also expected to be replaced, but not. The refactored program in Listing 11 could be compiled successfully, but the refactoring result does not align with the user's intention, as the users would like all occurrences of the expression to be refactored correctly. Figure 12 also gives an example[23] in which the rename refactoring is partially completed. When a user renames the foo variable in line 5 of the Listing 12, IntelliJ IDEA automatically updates the setter method except for the parameter name in the interface method in line 2 of Listing 13. Figure 13 shows an example that IntelliJ IDEA makes no change when performing inline class refactoring. For the input program in Listing 14, the inline class refactoring is available in the class Action, when performing this refactoring, users would expect the refactored program in Listing 15. However, as stated in the bug report[24], nothing changed after applying the inline class refactoring.

Comment Related (22). The refactoring engine successfully applies the refactoring, however, the refactoring engine does not handle the corresponding code comment appropriately. For example[25] as shown in Figure 14, invoking "Safe Delete" on the method parameter "arg2" in line 1 in Listing 16 in IntelliJ IDEA would result in the refactored program in Listing 17. The Javadoc on line 2 of Listing 17 is broken due to the unmatched parameter type. The correct Javadoc should be "@link #abc(Integer, Integer)" instead of "@link #abc(String, String)" because the two parameters (i.e., "arg1" and "arg3") remained in method "abc()" at line 1 are both "Integer".

[23] https://youtrack.jetbrains.com/issue/IDEA-146229
[24] https://youtrack.jetbrains.com/issue/IDEA-149230
[25] https://youtrack.jetbrains.com/issue/IDEA-140384

```
import static java.util.Arrays.*;
class Bug {
    {
        String[]side=new String[0];
        if(true){
-           System.out.println(asList(side));
        }
        else{
            System.out.println(asList(side));
        }
    }
}
```

Listing 10. Code before refactoring.

```
+import java.util.List;
import static java.util.Arrays.*;
class Bug {
    {
        String[]side=new String[0];
+        List<String> list = asList(side);
        if(true){
+           System.out.println(list);
        }
        else{
            System.out.println(asList(side)); //
                should be refactored
        }
    }
}
```

Listing 11. Code after refactoring.

Fig. 11. Eclipse-104293: extract local does not replace all concurrences of expression.

```
interface Foo {
    void setFoo(int foo);
}
class Bar implements Foo {
-    int foo;
    @Override
-    public void setFoo(int foo) {
-        this.foo = foo;
    }
}
```

Listing 12. Code before refactoring.

```
interface Foo {
    void setBar(int foo);
}
class Bar implements Foo {
+    int bar;
    @Override
+    public void setBar(int bar) {
+        this.bar = bar;
    }
}
```

Listing 13. Code after refactoring.

Fig. 12. IDEA-146229: parameter name in interface method is not renamed.

```
import java.io.Serializable;
- class Action implements Serializable {}
class MyAction extends Action {
    <T extends Action> void m() {}
}
```

Listing 14. Input program.

```
import java.io.Serializable;
+ class MyAction implements Serializable {
    <T extends MyAction> void m() {}
}
```

Listing 15. Expected refactored program.

Fig. 13. IDEA-149230: IDEA makes no change when inline super class in method with generic restriction.

```
public void abc(Integer arg1, String arg2, Integer
      arg3) {}
/** {@link #abc(Integer, String, Integer)} */
public void def() {}
```

Listing 16. Code before refactoring.

```
public void abc(Integer arg1, Integer arg3) {}
/** {@link #abc(String, String)} */
public void def() {}
```

Listing 17. Code after refactoring.

Fig. 14. IDEA-140384: Javadoc @link is broken after refactoring.

Unnecessary Change (17). The refactoring engine successfully applies the corresponding refactoring, however, the refactored programs contain some redundant changes. This happens because the refactoring engine developers are over-cautious, thus they make verbose changes (adds code that is not needed) to ensure the refactored program will not introduce errors. Even though those redundant changes will not introduce a compilation error or change the program behavior, the refactoring result is not aligned with the developer's expectations. For example, as shown in Figure 15, applying extract method refactoring for the statement at line 3 in Listing 18 will produce a refactored program that

```
1 ...
2 int a = 1;
3 System.out.println("" + (Object)a);
4 ...
```

Listing 18. Code before refactoring.

```
1 ...
2 int a = 1;
3 test(a);
4 ...
5 private void test(Object a) {
6     System.out.println("" + (Object)a);
7 }
```

Listing 19. Code after refactoring.

Fig. 15. IDEA-79743: extract method refactoring produces redundant cast.

```
1  public static void main(String[] args) {
2      System.out.println(abc());
3  }
4  public static String abc() {
5      return """
6             abc
7             defg
8             lksjkljsd
9             """;
10 }
```

Listing 20. Input program.

Fig. 16. Eclipse-551002: extract local variable and extract constant not available on text block.

```
1  public class MethodTest{
2      public void methodOne() {
3          System.err.println("one");
4      }
5      public void methodTwo() {
6          System.err.println("two");
7      }
8      public void methodThree() {// rename this method
9          System.err.println("three");
10     }
11 }
```

Listing 21. Input program.

Fig. 17. Eclipse-280518: method breakpoints all change to one method after rename method refactoring.

contains redundant type casts. As shown in Listing 19, the parameter "a" is an "Object" type, however, it is casted to "Object" again at line 6. The developer complains that "cast a to Object is redundant"[26].

Refactoring Not Available (12). The refactoring operation is not available on some program elements (e.g., text block and annotation) even though it is supposed to be. For example[27], as shown in Figure 16, the developer complains that "Extract Local Variable", "Extract Constant" and "Inline" refactoring are not available in the "abc()" method for the input program in Listing 20 because it contains a text block between line 5 and line 9.

Others (6). The remaining bugs exhibit other symptoms, e.g., broken breakpoints and poor performance. For example, developers first placed method breakpoints in "methodOne()", "methodTwo()", and "methodThree()" for the input program in Listing 21 of Figure 17. Then, they selected "methodThree()" and changed its name to "methodFour()" using the rename method refactoring. They ended up with all the three breakpoint markers changed to "methodFour()"[28].

[26] https://youtrack.jetbrains.com/issue/IDEA-79743
[27] https://bugs.eclipse.org/bugs/show_bug.cgi?id=551002
[28] https://bugs.eclipse.org/bugs/show_bug.cgi?id=280518

Table 5. Bug distribution by symptoms.

Symptom	Eclipse	IntelliJ IDEA	NetBeans	Total
Compile Error	88	114	40	242
Crash	82	14	10	106
Behavior Change	23	33	10	66
Failed Refactoring	9	10	5	24
Incorrect Warning Message	9	14	0	23
Comment Related	7	12	3	22
Unnecessary Change	7	9	1	17
Refactoring Not Available	9	3	0	12
Others	6	0	0	6

Table 5 presents the distribution of refactoring engine bugs according to the symptom categories. It shows that "Compile Error" is the most common symptom in all three refactoring engines. It occurs when refactoring engines generate refactored programs containing syntax errors, accounting for 88 Eclipse bugs, 114 IntelliJ IDEA bugs, and 40 Netbeans bugs, respectively. Determining the syntactic correctness of a refactored program is not difficult with the help of JVM compilers. Therefore, the developer could test such bugs by compiling the refactored program and collecting the error messages to fix the syntax errors. "Crash" is the second most common symptom. 82, 14, and 10 bugs exhibit this symptom in Eclipse, IntelliJ IDEA, and Netbeans, respectively. As detection of crashes does not require explicit test oracles, the large percentage of crashes suggests the potential of augmenting the existing test suite with the generated or mutated ones. Besides, engine developers can design effective localization and deduplication methods based on the stack trace information collected from crashes to locate the root causes of the crashes. We did not classify the above two symptoms into more fine-grained subcategories because they will not give us very useful information. For example, the symptom of compile error is caused by syntax errors in the refactored program, and the symptom of crash indicates that the refactoring engine terminates unexpectedly with exception. Classifying syntax errors into finer categories like "cannot find symbol" will not give us much insight, since those syntax errors are not previously unknown and are rather shallow. For the crash, exceptions like "NullPointerException", "InvocationTargetException", or "ClassCastException" will also not give us very useful takeaways. "Behavior Change" occurs when refactoring engines generate non-behavior-preserving refactored code. It is the third common symptom as shown in Table 5, taking up 66 bugs in total. This symptom is not as obvious as "Compile Error" and "Crash". Specifically, the refactored program contains no syntax errors, determining the behavior-preserving of a refactoring is hard due to its complexity. Existing testing tools based on SAFEREFACTOR [46, 64–68] try to identify those kind of bugs by randomly generating tests, however, they can only generate test cases for the common methods before and after the refactoring in the project, which means they cannot exercise the changed methods (e.g., refactored methods) thoroughly. Besides, according to a previous study [62], automatic test generation tools (e.g., Randoop and EvoSuite) are not effective for refactoring validation and they could miss more than half of injected refactoring faults. Therefore, testing such bugs is challenging because the test oracles are difficult to define. Further, the adverse impact of this category of bugs is severe since it violates the behavior-preserving assumption of a refactoring engine. Hence, our study indicates that it is worthwhile for future research in testing refactoring engines to focus on designing effective techniques for identifying non-behavior-preserving refactoring bugs.

Finding 2: "Compile Error" (242), "Crash" (106), and "Behavior Change" (66) are the top three most common symptoms of refactoring engine bugs.

According to Table 5, the symptoms of "Failed Refactoring" (24), "Incorrect Warning Message" (23), and "Comment Related" (22) are not negligible. "Failed Refactoring" could be partly detected by checking whether the original program has been changed or not, but for partially completed refactoring, it is hard to detect the incomplete change, since it still compiles successfully without any syntax error. Detection of "Incorrect Warning Message" and "Comment Related" is also difficult since there are no standard oracles, but with multiple refactoring engines, differential testing that has been used in testing static analysis tools [106] could be used to compare the refactoring results to find defects. Similarly, testing "Unnecessary Change" (17) is challenging because the test oracles are difficult to define (the question of "Which change is redundant?" needs to be answered to design the oracle for each type of refactoring), and one possible solution is to use the refactoring results of multiple refactoring engines to identify the redundant change. Meanwhile, "Refactoring Not Available" (12) could be detected by checking the availability of the refactoring type for the input programs that satisfy the preconditions. Meanwhile, we classify some symptoms into "Others" since their bug number is small, like the "Bad performance" (2) and "Broken Breakpoints" (3). The "Broken Breakpoints" together with the "Comment Related" symptoms indicate that developers of refactoring engines should take into account the various side effects of the refactoring operations because maintaining the consistency between the refactored code and other software artifacts (e.g., code comments and breakpoints) is as important as preserving the functionality of the original program.

Finding 3: Apart from the refactored code itself, other aspects such as warning messages and the availability of the refactoring could also be error-prone. Current refactoring engines have not thoroughly taken into account various side effects that may be incurred by the refactoring. It is important to maintain consistency between the refactored code and other software artifacts (e.g., code comments and breakpoints).

4.3 RQ3: Root Causes

Based on the classification and labeling process described in Section 3.2, all the root causes compiled of the refactoring engine bugs are presented as follows.

Incorrect Transformations (109). The most common root cause is incorrect transformations, which happens when refactoring engines try to rewrite the AST and create code modifications. Since the transformations in the refactoring engines are rule-based, manually crafted, and heavily relying on the refactoring engine developers' expertise, it is natural to be error-prone. In addition, engine developers are unaware of all possible scenarios for each refactoring considering the diversity and grammar complexity of the input programs. To better understand, we further classify the current categories into fine-grained subcategories. During the classification process, we first obtain the input program, the desired refactored program, the actual refactored program, the discussions, and the corresponding patch for each historical bug report. Then, we compare the desired refactored program with the actual refactored program to locate the incorrect part. By analyzing the patch and the discussions for each bug report, we summarize the root cause in natural language. Then, we follow the open-coding scheme described in Section 3.2 to develop the taxonomies. In some historical bug reports, the desired refactored program and the actual refactored program are not available, so we filter these out. In total, we build our subcategories based on 55 of 109 historical bug reports from the current categories. Specifically, a detailed description for each subcategory is given below.

- **Improper handling code comments (15).** The code comments are not properly handled during the transformation, leading to deleted, outdated, or incorrect code comments. For example[29], Listing 22 in Figure 18 gives

[29] https://bugs.eclipse.org/bugs/show_bug.cgi?id=305103

```
1  class A {
2  -   /*
3  -   * A very important comment.
4  -   */
5  -   class X {}
6  }
```

Listing 22. Code before refactoring.

```
1  class A {}
2  + class X {}
```

Listing 23. Code after refactoring.

Fig. 18. Eclipse-305103: comment deleted on "Move type to new file" refactoring.

```
1  class OriginalClass {
2       private boolean flag = false;
3  -    public synchronized void originalMethod(){
4            // Some logic here
5            flag = true;
6            notify();
7       }
8  -    public void callerMethod(){
9  -        originalMethod();
10 -    }
11 }
```

Listing 24. Code before refactoring.

```
1  class OriginalClass {
2       private boolean flag = false;
3  +    public void callerMethod(){
4            // Some logic here
5            flag = true;
6            notify();
7       }
8  }
```

Listing 25. Code after refactoring.

Fig. 19. Eclipse-1360: inline method refactoring leads to changes in access permissions.

an input program. There is an inner class X together with a multi-line comment inside class A. After applying "Move type to new file" refactoring on class X in Eclipse 3.6, it is moved to a new Java file as a standalone class as shown in Listing 23, however, the comment exists neither in the original class A nor in the moved class X. This happens because Eclipse does not handle code comments properly during rewrite the AST of the input program. As the refactoring engine developer stated, the code comments should remain in the original class or move together with class X.

- **Incorrect modifier modification (14).** In Java, a modifier[30] is a keyword used to change or specify the behavior, visibility, or properties of classes, methods, variables, and other elements. Modifiers are typically divided into two categories: (1) Access modifiers (i.e., public, protected, default, and private) specify who can access a class, method, or variable; (2) Non-Access modifiers (e.g., static, final, abstract, synchronized, and volatile) specify how code behaves. During transformation, incorrect modifier modification could result in either compiler error or behavior change. For example, in Figure 19, when a user performs "Inline Method" refactoring for `originalMethod()` in line 10 for the input program in Listing 24 using Eclipse version 2023-09, the keyword "synchronized" is lost in the method declaration of the refactored code as shown in the refactored program in Listing 25, resulting in a behavior change. As the user stated: "Before refactoring, inline methods contain access permissions for synchronization modifiers but are missing after inlining."[31] By analyzing the patches[32] together with the discussions, we concluded that this happens because engine developers ignored the case where methods own a "synchronized" modifier when rewriting AST, thus there is no code logic to process the synchronized block when performing inline method refactoring.
- **Unnecessary change (11).** As we discussed in Section 4.2, unnecessary change indicates that refactoring engine tends to add verbose changes (i.e., code that is redundant) to ensure the refactored program will not introduce

[30] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Modifier.html
[31] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1360
[32] https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/1378

```
1 class X {
2 -  String string() {
3 -    return "heavy" + "lighting" + "metal";
4 -  }
5    String x() {
6      return "start" + string() + "end";
7    }
8 }
```

Listing 26. Code before refactoring.

```
1 class X {
2     String x() {
3 +   return "start" + ("heavy" +
4 +         "lighting" + "metal") + "end";
5     }
6 }
```

Listing 27. Code after refactoring.

Fig. 20. IDEA-314882: inline method adds unnecessary parentheses when inlining a string concatenation into a string concatenation.

```
1 public class CallToSimpleGetter {
2     private String someString = "";
3     public void execute() {
4 -       getFirst();
5     }
6     public String getFirst() {
7         return someString;
8     }
9 }
```

Listing 28. Code before refactoring.

```
1 public class CallToSimpleGetter {
2     private String someString = "";
3     public void execute() {
4 +       someString;
5     }
6     public String getFirst() {
7         return someString;
8     }
9 }
```

Listing 29. Code after refactoring.

Fig. 21. IDEA-127135: inline call to getter generates invalid code.

```
1 class Base {
2    public void foo(int count) {}
3 }
4 class A extends Base {
5     @Override
6 -   public void foo(int count) {}
7 }
```

Listing 30. Code before refactoring.

```
1 class Base {
2    public void foo(int count) {}
3 }
4 class A extends Base {
5     @Override
6 +   public void foo1(int count) {}
7 }
```

Listing 31. Code after refactoring.

Fig. 22. IDEA-154669: change signature should remove @Override when refactor only current method.

errors. This could be attributed to the developers of the refactoring engine being overly cautious, even though the outcome is not in line with the developer's expectations. Figure 20 shows an example[33] in which IntelliJ IDEA 2023.2 adds unnecessary parentheses when performing an inline method refactoring for the input program in Listing 26. As shown in the Listing 27, the returned string is the same with or without parentheses, making them unnecessary.

- **Invalid statement (8).** In the refactored program, there could exist invalid statements due to the improper code transformation. For example[34], using IntelliJ IDEA to perform inline method refactoring for the method call on line 4 of Listing 28 in Figure 21 would generate an invalid statement. As shown in the Listing 29, an invalid statement is created in line 4 which could result in a syntax error.
- **Incorrect annotation handling (4).** Some refactoring could affect the usage of existing annotations in the input program. For example[35], as shown in Figure 22, the method foo() in line 6 of Listing 30 is changed to foo1() as shown in Listing 31, however, IntelliJ IDEA would leave the annotation @Override unchanged. There is no method named foo1() in the super class Base, so the original annotation @Override will result in a syntax error.

[33] https://youtrack.jetbrains.com/issue/IDEA-314882
[34] https://youtrack.jetbrains.com/issue/IDEA-127135
[35] https://youtrack.jetbrains.com/issue/IDEA-154669

```
1  class C {
2    void m() throws Exception {
3  -     try (AutoCloseable inlineMe = null) {
4  -       try (AutoCloseable r2 = inlineMe) {
5  -         System.out.println(inlineMe + ", " + r2);
6        }
7      }
8    }
9  }
```

Listing 32. Code before refactoring.

```
1  class C {
2    void m() throws Exception {
3  +     try {
4  +       try (AutoCloseable r2 = null) {
5  +         System.out.println(r2 + ", " + r2);
6        }
7      }
8    }
9  }
```

Listing 33. Code after refactoring.

Fig. 23. IDEA-109541: inline resource variable produces uncompilable code.

```
1  public class Test {
2      public void test() {
3          int a = 1;
4          if (true) {
5              System.out.println(a);
6          } else {
7  -           System.out.println(a);
8          }
9      }
10 }
```

Listing 34. Code before refactoring.

```
1  public class Test {
2      public void test() {
3          int a = 1;
4          if (true) {
5              System.out.println(a);
6          } else {
7  +           foo();
8          }
9      }
10 +    private void foo() {
11 +        System.out.println(a);//syntax error
12 +    }
13 }
```

Listing 35. Code after refactoring.

Fig. 24. IDEA-87654: IntelliJ IDEA fails to detect input variable if code fragment is located in unreachable else branch, the extracted method contains syntax error.

- **Incorrect exception handling (3).** Refactoring engines face challenges when the exception handling code is refactored. For example[36], when performing an inline variable refactoring for the resource declaration on line 3 of Listing 32 in Figure 23 using IntelliJ IDEA, the variable is correctly inlined, however, the try-with-resources in the input program is not correctly transformed. The try statement on line 3 of Listing 33 is not closed with catch, finally, or resource declarations, thus resulting in a syntax error.

Incorrect Flow Analysis (97). Incorrect flow analysis in refactoring engines is a common root cause of a bug in refactoring that depends on the control flow of the code and the data flow. Refactoring like the "Extract Method" often relies heavily on accurate flow analysis to identify variables, control structures, and dependencies correctly, when the flow analysis is flawed or not in the right scope, it can lead to error in the refactoring results. Incorrect flow analysis could result in incorrect reference update, incorrect import, etc. For example, in Figure 24[37], for the input program in Listing 34, IntelliJ IDEA fails to detect input variables when the code fragment to be extracted is located in an unreachable else branch, resulting in an uncompilable refactored code in Listing 35. This issue occurs because IntelliJ IDEA skips the unreachable code during control flow analysis for the "Extract Method" refactoring[38].

Incorrect Preconditions Checking (62). Each refactoring contains some preconditions to guarantee that the refactored program will be behavior-preserving or syntactically correct. For example, when implementing Make Static refactoring, the Eclipse developers propose a list of preconditions that must be checked[39] (e.g., the method to be made static should not be a constructor method). In practice, testing refactoring preconditions involves manually creating an input program

[36] https://youtrack.jetbrains.com/issue/IDEA-109541
[37] https://youtrack.jetbrains.com/issue/IDEA-87654
[38] https://github.com/JetBrains/intellij-community/commit/961086a6d1dc2882c38c4b752267db41aec91816
[39] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/590

```
1 -record R(int i) {
2         R {
3                 System.out.println("Compact
                        Constructor");
4         }
5 -}
6 -class X {}
```

Listing 36. Code before refactoring.

```
1 +record R(int i) {}
2 +class X {
3         R {
4                 System.out.println("Compact
                        Constructor");
5         }
6 +}
```

Listing 37. Code after refactoring.

Fig. 25. Eclipse-566982: move operation should not be allowed for Constructors.

```
1 -public interface I {}
2 -public class C implements I {
3 -    boolean value;
4 -}
```

Listing 38. Code before refactoring.

```
1 +public interface I {
2 +    boolean value;
3 +}
4 +public class C implements I {}
```

Listing 39. Code after refactoring.

Fig. 26. Eclipse-131348: pull up refactoring allows pulling fields into an interface.

to be refactored and specifying a refactoring precondition failure as expected output. However, developers choose input programs for just checking the preconditions they are aware of. Since specifying preconditions is a nontrivial task, developers may be unaware of the preconditions needed to guarantee behavioral preservation or syntax correctness.

- **Overly weak preconditions (54).** When the implemented preconditions are insufficient to guarantee behavior preservation or syntactic correctness, we call it "Overly Weak Preconditions", which may introduce compilation errors or behavioral changes. Figure 25 gives an example[40] where a move method refactoring is allowed for the constructor of a class, which results in a syntax error. As shown in the Listing 36, it is able to perform refactoring of the move method for the constructor method of the record class R in line 2. After moving the constructor to class X, the refactored program in Listing 37 contains a syntax error because the moved method is not syntactically correct in class X. Figure 26 also gives an example[41] where a variable is allowed to be pulled to the interface of the input program in the Listing 38. As shown in the Listing 39, in the refactored program, there is an interface named I that contains a variable, which is against the Java language specifications. Pulling up a non-static variable into an interface should not be allowed at the precondition checking stage of pull up refactoring.
- **Overly strong preconditions (8).** Some implemented preconditions may be overly strict because the refactoring engine developers are over-cautious. We consider them as "Overly Strong Preconditions", and could result in the engine preventing developers from applying an applicable refactoring. Figure 27 shows an example[42] where a move method refactoring is not available for the method in line 5 of the input program in Listing 40 due to overly strong preconditions. As the user complained — "The inner class is non-static, but the method only accesses a static field. So Eclipse should not refuse to move it. Interestingly, Eclipse allows moving the number field, but not the method, even though they are in the same inner class."

Incorrect Type Resolving (45). Bugs in this category are caused by incorrect type-related operations such as type inference, type binding, type matching, etc. For instance, type binding refers to the process of resolving references in Java source code to their corresponding types. Specifically, a type binding associates variable declarations, method

[40] https://bugs.eclipse.org/bugs/show_bug.cgi?id=566982
[41] https://bugs.eclipse.org/bugs/show_bug.cgi?id=131348
[42] https://bugs.eclipse.org/bugs/show_bug.cgi?id=107998

```
1 public class Bug {
2     public static int field;
3     class Inner{
4         int number=field;
5         int method(){return field;}
6     }
7 }
```

Listing 40. Input program.

Fig. 27. Eclipse-107998: move method out of non-static inner class too strict.

```
1 public class A {
2     void foo(Object x) {
3         if (x instanceof String) x = ((String)x).
              substring(1);
4         if (x instanceof String) x = <selection>((
              String)x).substring(1)</selection>;
5     }
6 }
```

Listing 41. Code before refactoring.

```
1 public class A {
2     void foo(Object x) {
3         if (x instanceof String) x = getSubstring(
              x); // syntax error
4         if (x instanceof String) x = getSubstring
              ((String) x);
5     }
6     private String getSubstring(String x) {
7         return ((String)x).substring(1);
8     }
9 }
```

Listing 42. Code after refactoring.

Fig. 28. IDEA-98396: extract method inserts casts in duplicated usages.

parameters, or return types with their concrete types. This information is stored in the AST's binding nodes for further operations. The bindings are crucial for understanding the program's structure and ensuring that the IDEs can perform accurate code analysis, such as renaming symbols. For example[43], as shown in Figure 28, extracting the code between <selection> and </selection> into a method in Listing 41 using IntelliJ IDEA will produce a refactored program with syntax error. For the refactored program in Listing 42, when calling the method getSubstring() on line 3, the variable x should be cast into String type as in line 4. However, IntelliJ IDEA fails to resolve this type cast during extract method refactoring.

Failed Selection Parsing (9). This category of bugs occurs when the refactoring engine fails to parse the input programs. Based on the discussions in the bug reports and patches, we further divide this category into three subcategories.

- **Unsupported new Java language features (4).** This happens because some features of the language in the latest version of JDK are not fully supported in IDEs. For example, in Eclipse bug report #156[44], a user reports that renaming refactoring does not work for "Record Patterns" [51], as the developers of the refactoring engine reply in the bug report: "Currently, there is no AST node for variables such as xy and lr. This should be revisited after Record Pattern DOM AST changes is merged".
- **Parsing for some program elements is not well-designed (3).** For example, IntelliJ IDEA users complain that some refactoring do not work for strings containing injected languages[45], or file paths[46]. The same situation also exists in Eclipse, as shown in Figure 29[47], when users try to extract a constant for a string that is the value of the annotation in the input program in Listing 43, the refactoring is not available in the pop-up because the annotation type is not implemented when parsing Java text selection to extract.

[43] https://youtrack.jetbrains.com/issue/IDEA-98396
[44] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/156
[45] https://youtrack.jetbrains.com/issue/IDEA-90322
[46] https://youtrack.jetbrains.com/issue/IDEA-97682
[47] https://bugs.eclipse.org/bugs/show_bug.cgi?id=303617

```
import java.util.ArrayList;
//@SuppressWarnings("all")
//@SuppressWarnings({"all", "rawtypes"})
//@SuppressWarnings(value= "all")
@SuppressWarnings(value= {"all", "rawtypes"})
public class Try extends ArrayList {
}
```

Listing 43. Input program.

Fig. 29. Eclipse-303617: refactoring is not available on pop-up if selected string is in an annotation in Eclipse 3.6.

Table 6. Bug distribution by root causes.

Root cause	Eclipse	IntelliJ IDEA	Total
Incorrect Transformations	51	58	109
Incorrect Flow Analysis	15	82	97
Incorrect Preconditions Checking	26	36	62
Incorrect Type Resolving	21	24	45
Failed Selection Parsing	6	3	9
Others	3	1	4

- **Inflexible parsing (2).** Some users complain that refactoring engines are not flexible when parsing the selected program elements to be refactored. For example, "Extract Method" refactoring does not work when a comment is selected at the end of the statements in IntelliJ IDEA[48]. In Eclipse 3.8, "Extract Method" refactoring would fail if the trailing comma is not selected for a statement[49]

Others (4). Each case in this root cause does not occur frequently and does not belong to any other root causes above. Including compatibility issues with the JDK version[50], performance issues caused by time-consuming method calls (e.g., the "toString()" method might cause a loss of performance[51]), and incorrectly getting user configuration[52].

Table 6 gives the distribution of bugs by their root causes. As shown in the table, "Incorrect Transformations" are the most common root cause. It accounts for 109 bugs in total, including 58 bugs in IntelliJ IDEA, and 51 bugs in Eclipse. This happens because refactoring engine developers need to write the code transformation rules manually depending on the input programs to be refactored. However, the input programs could be diverse due to the complex grammar and different usage scenarios, thus developers might not be aware of the input program beyond their knowledge, as a result, the designed rules could be flawed. On the other hand, developers could make mistakes even with the input programs since writing transformation rules manually is non-trivial. The result indicates that handling transformation issues in refactoring engines is very challenging and deserves more attention. Based on our study, we conclude two directions that developers and researchers could focus on to tackle this problem:

- **Towards obtaining diverse input programs.** Diverse input programs could help developers from two aspects: First, those input programs could be used to design unit tests to expose bugs in refactoring engines; Second, with the help of diverse input programs, developers could design more robust program transformation rules considering different refactoring scenarios. The diverse input programs could be obtained from real-world

[48] https://youtrack.jetbrains.com/issue/IDEA-91985
[49] https://bugs.eclipse.org/bugs/show_bug.cgi?id=366281
[50] https://bugs.eclipse.org/bugs/show_bug.cgi?id=547094
[51] https://bugs.eclipse.org/bugs/show_bug.cgi?id=570587
[52] https://youtrack.jetbrains.com/issue/IDEA-134349

projects with the help of automated refactoring detection tools (e.g., RefactoringMiner [97]). Alternatively, since large language models (LLMs) such as ChatGPT trained on massive data have achieved impressive results in many software engineering tasks, leveraging LLMs to generate diverse input programs could be promising. Historical refactoring engine bug reports or input programs in existing test suites could be used to construct the prompts to generate more diverse input programs.

- **Automate the code refactoring in a data-driven way.** Previous studies [36, 40–42] have tried to automate code transformations in a data-driven way. They have obtained comparable results; besides, more diverse transformations could be applied since they are trained on massive data from real-world projects. Recently, IntelliJ IDEA has introduced AI code assistants to help users automate code-related tasks, including suggesting refactoring [88]. There could be potential to explore code refactoring automation in the era of LLMs.

Finding 4: Our study found that "Incorrect Transformations" (109) is the most common root cause for Eclipse and IntelliJ IDEA, accounting for 51 and 58 bugs, respectively. We conclude two directions as resolutions: (1) Towards obtaining more diverse input programs; (2) Automate the code refactoring in a data-driven way.

The bugs caused by "Incorrect Flow Analysis" and "Incorrect Type Resolving" are also non-negligible, accounting for 97 bugs and 45 bugs, respectively. Flow analysis is fundamental for refactoring, ineffective flow analysis could result in behavior change, failed refactoring, and uncompilable code. A previous study [8] also has shown that "Extract Local Variable" refactoring in Eclipse and IntelliJ IDEA could lead to behavior change due to inaccurate flow analysis and ineffective identification of side effects. "Incorrect Type Resolving" (i.e., incorrect handling of type-related operations), such as type inference, type binding, and type matching, is a common root cause of bugs. These issues often arise because refactoring involves complex code transformations that depend on correctly manipulating the types of variables, expressions, and methods. For example, during refactoring, especially in Java versions that support features such as the "var" keyword, refactoring may fail due to incorrectly inferred types. For example, Eclipse 4.11 fails to convert a local variable to a field in the "var" variable because it does not resolve to its inferred type during refactoring [89]. Meanwhile, "Failed Selection Parsing" (9) caused by some rare input programs, unsupported language features, and inflexible parsing should also be paid attention. Except for the above-mentioned root causes, refactoring engines might be affected by the different versions of JDK, or slowed down by some time-consuming method calls.

Finding 5: "Incorrect Flow Analysis" (97) and "Incorrect Type Resolving" (45) are also non-negligible root causes. Refactoring engine developers should take into account different scenarios for parsing input programs.

"Incorrect Preconditions Checking" is the third most common root cause, accounting for 62 bugs, including 54 bugs caused by "Overly Weak Preconditions" and 8 bugs caused by "Overly Strong Preconditions". Existing studies [47, 67] focus mainly on revealing the bugs caused by "Incorrect Preconditions Checking", however, as shown in Figure 6, it only covers a small portion of bugs. Besides, manually crafting preconditions for each refactoring could be non-trivial since developers need to consider different scenarios of input programs. Meanwhile, with the introduction of new language features in the newer version of JDK, each refactoring precondition needs to be updated accordingly. There are works [3, 9, 30, 54] that try to infer preconditions for debugging or program verification through symbolic analysis or data-driven way. However, the precondition inference for refactoring has not been well explored. Given the input program and existing preconditions, new preconditions should be predicted automatically.

Finding 6: "Incorrect Preconditions Checking" (62) is the third most common root cause. Preconditions need to be updated considering the introduction of new language features. Considering the evolution of the programming language, automating the inference of preconditions for refactoring could be a promising research direction.

4.4 RQ4: Triggering Conditions

We check the discussions and procedures for reproducing the bugs in each bug report. If a user does not explicitly mention changing the default refactoring input options, we consider the bug to be triggered by the default configuration. Overall, only 15 (2.9%) bugs are not triggered by the default configuration; in those cases, users need to select extra options in the refactoring user interfaces (e.g., when extract method in IntelliJ IDEA, the user could check the "Declare static" to make the extracted method a static method). This means that the rest of the refactoring engine bugs (97.1%) could be triggered by the default initial input options. This is a useful finding, since configuring the input options while testing refactoring engines could increase search space and is time-consuming. For example, while using fuzzing testing to generate input programs to uncover the bugs, we could just use the default input options without considering the configuration-related bugs, which could be faster and more efficient.

Finding 7: Most of the bugs (97.1%) could be triggered by the default initial input options of the refactoring engines.

For the RefactorBench dataset, we manually check the input program and categorize its characteristics. This step aims to identify the input program characteristics that are more prone to trigger refactoring engine bugs. The purpose of the current research question is twofold: 1) By identifying the input program characteristics and features, the researchers and engine developers could use that information to guide the test case generation techniques to produce more error-prone input programs, improving the effectiveness and efficiency of testing. For example, researchers could use our findings in Table 7 and seed input programs from our dataset to construct prompts and instruct LLM (e.g., ChatGPT) to generate more diverse input programs (refer to Section 6.1.1 for more details). In this way, it could potentially trigger more bugs in the refactoring engines. 2) Engine developers could design more diverse and complete unit tests for each refactoring type by referring to our findings, improving the robustness and usability of IDEs.

Table 7 lists the identified taxonomy of the characteristics of the input program. We derived our taxonomy based on the RefactorBench dataset, which includes 518 historical bug reports. However, during our labeling process, we found that some input programs do not have explicit features that can lead to safe conclusions. After filtering out these types of bug reports, only 270 bug reports remained (124 from Eclipse, 121 from IntelliJ IDEA, and 25 from Netbeans). As a result, we derived our taxonomy based on these remaining bug reports. The "Category" column in Table 7 describes the high-level types of program characteristics, while the "Sub-category" column gives the specific categories. The last column (Total (#/%)) presents the total number and percentage of reports that fit into a particular category. In total, we identified eight main categories with 38 subcategories. Among the eight categories, we observed that input programs that involve Java language features are the most likely to trigger refactoring engine bugs during refactoring activities. For example, refactoring with a lambda expression in the input program could trigger 39 bugs, since code transformations for a lambda expression usually involve type inference and complex AST rewrite. This category represents more than half (64.4%) of the bug reports studied. Existing studies [10, 21, 46, 64–68] to test the refactoring engines do not know about the bug-triggering ability of the input programs. ASTGen [10] relies on manually defined templates to generate input programs, defining the templates requires expertise, and developers do not know how to design bug-triggering-prone templates, which could result in low efficiency and low effectiveness while testing.

Table 7. Categories of bug-triggering input program characteristics in our studied bug reports.

Category	Sub-category	Description	Bug (#)	Total (#/%)
(T1) Language Features	(T1.1) Lambda expression	Anonymous functions used to implement functional interfaces with a more streamlined syntax	**39**	**174/64.4**
	(T1.2) Java generics	Java generics allow to create classes, interfaces, and methods that operate with unspecified types	**38**	
	(T1.3) Enum	A special data type used to define a fixed set of constants representing a set of predefined values	**27**	
	(T1.4) Record	A special kind of class that helps encapsulating related data, introduced in Java 14	13	
	(T1.5) Varargs	It allows a method to accept a variable number of arguments of the same type	11	
	(T1.6) instanceof	It is a Java keyword used to check if an object is an instance of a particular class or interface	8	
	(T1.7) Foreach	A concise loop construct used to iterate over elements in an array or collection without explicit indexing	8	
	(T1.8) Switch case	It provides a concise way to conditionally execute blocks of code based on the value of a variable	8	
	(T1.9) Try-with-resources	It can that automatically manages resources, ensuring they are closed at the end of a block of code	5	
	(T1.10) Var	Enables implicit type inference, allowing for more concise code while maintaining strong typing	4	
	(T1.11) Try-catch-finally	Java try-catch-finally is a mechanism used for handling exceptions	3	
	(T1.12) Joint variable/field declaration	It allows multiple variables or fields of the same type to be declared in a single statement	3	
	(T1.13) Multi-dimension Array	A data structure that organizes elements in multiple dimensions	2	
	(T1.14) Vector	Vector is a synchronized, resizable array implementation of the List interface	2	
	(T1.15) Synchronized block	It ensures that only one thread at a time can execute the code within the block	1	
	(T1.16) Java ternary conditional	A concise way to write conditional statements, returning one of two values based on a specified condition	1	
	(T1.17) Keyword "this"	The "this" keyword in Java refers to the current instance of the class, allowing access to its own members	1	
(T2) Class-related	(T2.1) Inner class	Class declared within another class or method, enabling tighter encapsulation and logical grouping of code	19	**35/13.0**
	(T2.2) Anonymous class	Class defined without a name, often used for one-time implementations of interfaces or abstract classes	15	
	(T2.3) Cyclically dependent class	Two or more classes depend on each other directly or indirectly	1	
(T3) Annotations	(T3.1) Annotations	Annotations can be used by the compiler or at runtime for various purposes such as configuration	29	**29/10.7**
(T4) Code Comment	(T4.1) Comment related	Related with Java comments, including comments in method and class	18	18/6.7
(T5) Method-related	(T5.1) Overloaded method	Methods within the same class that share the same name but have different parameter lists	6	12/4.4
	(T5.2) Static method	A method that belongs to the class rather than any specific instance	2	
	(T5.3) Method reference	A shorthand syntax for referring to methods by their names, enabling concise and readable code	2	
	(T5.4) Recursive method	A function that calls itself within its definition	1	
	(T5.5) Default method	Method defined within an interface with a default implementation	1	
(T6) Static	(T6.1) Static initializer	It is typically used for initializing static fields or performing one-time setup tasks	4	7/2.6
	(T6.2) Static import	Fields and methods defined in one class to be referenced in another class without specifying the class name	2	
	(T6.3) Static field	A class-level variable shared among all instances of the class	1	
(T7) Constructor-related	(T7.1) Super constructor	It is used to initialize the superclass of an object	4	6/2.2
	(T7.2) Nested constructor	Refers to a constructor within a class that can be invoked by another constructor in the same class	1	
	(T7.3) Implicit constructor	An implicit constructor is automatically provided by the compiler when no explicit constructor is defined	1	
(T8) Others	(T8.1) Special String	Like injected languages, string contain special tokens	6	11/4.1
	(T8.2) Arithmetic expression	Mathematical operations performed on numerical values using operators like +, -, *, /, and %	2	
	(T8.3) Time-consuming method call	For example, calling toString() method in a instance involving resource access would be time-consuming	1	
	(T8.4) Dead code block	A block of code that is never executed	1	
	(T8.5) Method chaining	A sequence of operations on an object without needing to store intermediate results in separate variables	1	

Complementarily, our work could serve as background information when designing templates, boosting effectiveness and efficacy. Gligoric et al. [21] apply some refactorings in all applicable program elements in real-world projects without considering the characteristics of bug-triggering input programs, with our findings, one direction to improve their work is to mine bug-triggering input programs from real-world projects and prioritize the testing on those programs. The same situation also happens to SAFEREFACTOR-based tools [46, 64–68], they test refactoring engines by generating unit tests to validate the behavior-preserving, however, refactoring should be validated on those error-prone input programs first. Existing studies [10, 21, 46, 64–68] to test the refactoring engines are unaware of the bug triggering ability of the input programs, are unguided, and need to explore a large search space before triggering bugs. Our findings could serve as complementary knowledge to improve their effectiveness and efficacy. Alternatively, our findings could also be used to design new techniques for testing refactoring engines. For example, researchers could use our findings in Table 7 and seed input programs from our data set to construct prompts and instruct LLM (e.g., ChatGPT) to generate more diverse input programs. In this way, it could potentially trigger more bugs in the refactoring engines. Based on our findings, more testing efforts should be made in input programs with language-specific features, considering 64.4% of our bug reports studied to be related to it, and its prevalence [13] in the OSS repositories.

Finding 8a: Refactoring programs involving Java language features are more likely to trigger bugs in refactoring engines, they take up 64.4% of our studied bug reports.

Among all the language features, input programs containing Lambda expressions (39/14.4%), Java generics (38/14.1%), and Enum (27/10.0%) are the three most prone to errors. Although refactoring engines like Eclipse and IntelliJ

```
default BitSet test(final List<T> items) {
    try {
        return ourPredicatesCache.get(Pair.create(this
                , items), () -> {
-       final BitSet result = new BitSet(items.size());
-       for (int i = 0; i < items.size(); i++) {
-         result.set(i, test(items.get(i)));
-       }
-       return result;
      });
    } catch (final ExecutionException e) {
      Logs.error(e);
      throw new RuntimeException(e);
    }
  }
```

Listing 44. Code before refactoring.

```
default BitSet test(final List<T> items) {
    try {
        return ourPredicatesCache.get(Pair.create(this
                , items), () -> {
+       return getBitSet();
      });
    } catch (final ExecutionException e) {
      Logs.error(e);
      throw new RuntimeException(e);
    }
  }
+ @NotNull
+ default BitSet getBitSet() {
+   final BitSet result = new BitSet(items.size());
+   for (int i = 0; i < items.size(); i++) {
+   result.set(i, test(items.get(i)));
+   }
+   return result;
+ }
```

Listing 45. Code after refactoring.

Fig. 30. IDEA-142163: extract method in IntelliJ IDEA fails to mark "items" as input argument resulting in compile error.

IDEA strive to provide robust support for the features of the modern Java language, there can still be instances where refactoring involving generics, Lambda expressions and Enum encounter difficulties due to the inherent complexity of these features and the limitations of refactoring engines. The main reasons are: (1) Lambda expressions and Java generics often involve type inference. It can be quite complex sometimes, especially in scenarios with nested generic types or where lambda expressions are involved. Refactoring tools need to accurately analyze and infer types to perform safe refactoring, and this complexity can lead to edge cases where the tools might not handle the refactoring correctly. For example, Figure 30[53] shows an example when extracting a method that contains a lambda expression for the input program in Listing 44, IntelliJ IDEA fails to mark the parameter "elements" as input arguments, resulting in a compile error for the refactored program in Listing 45. This is because when performing control flow analysis, code fragments where the parent nodes are lambda-type are not included, resulting in missing parameters. (2) Enums in Java are often used in ways that involve complex-type hierarchies or switch statements. Types of refactoring that involve Enum need to handle these cases correctly to ensure that the behavior of the code remains consistent after the refactoring. Enum refactoring also involves handling references to Enum constants and ensuring that they are updated correctly throughout the code base. (3) Refactoring tools typically analyze a limited scope of the codebase to ensure reasonable performance. However, this limited scope can sometimes miss relevant dependencies or interactions that are crucial for correctly performing the refactoring, especially when dealing with features like lambda expressions and generics that involve complex type relationships.

Finding 8b: Lambda expression (39/14.4%), Java generics (38/14.1%), and Enum (27/10.0%) are the top three language features that trigger refactoring engine bugs due to the complexity of type inference, limited flow analysis, and complicated usage scenario.

Input programs with complex class relationships are the second most bug-triggering category. In this category, the inner class (19/7.0%) and anonymous class (15/5.6%) are more likely to induce bugs in the refactoring engines. By analyzing the bug reports, input program, and usage scenario, we identified the following reasons: (1) Scope and visibility: Inner classes and anonymous classes have unique scope and visibility rules compared to top-level classes.

[53] https://youtrack.jetbrains.com/issue/IDEA-142163

```
1  public class ChangeMethodSignatureBug {
2      public ChangeMethodSignatureBug(Object obj) {
3      }
4      public void m() {
5          new ChangeMethodSignatureBug(new Object() {
6              public void a(Object par1, Object par2) {
7              }
8          });
9      }
10 }
```

Listing 46. Input program.

Fig. 31. Eclipse-393829: exception occurs when applying "Change Method Signature" on a method of an anonymous class in Eclipse.

They can access both the final local variables of the enclosing block and members of their enclosing class, which might not be static. Managing these visibility and scope distinctions during refactoring, particularly when moving classes or changing accessibility, can complicate the refactoring logic. As shown in Figure 31[54], for the input program in Listing 46, when changing the method signature for a method inside an anonymous class, an exception occurs. After analyzing the patch, we find that this is because when visiting the AST node for AnonymousClassDeclaration, it incorrectly stopped, resulting in an exception for the refactoring. (2) Complex naming and referencing: Anonymous classes, by their nature, lack a named class declaration, which makes them tricky in refactoring scenarios that involve class renaming or moving. Refactoring engines must handle references that are not tied to a simple class name but rather to their creation context, which can be nested deeply within methods or other classes. (3) Life-cycle and initialization concerns: Refactoring that changes initialization order, such as moving an inner class outside or converting it to a static nested class, can inadvertently alter when and how instances of the class are created. This can lead to subtle bugs if the initialization sequence or the dependencies on the enclosing instance are not carefully managed.

Finding 9: Input programs having complex class relationships are more likely to result in refactoring engine failure. Among these, refactoring involving inner class (19/7.0%) and anonymous class (15/5.6%) are the top two most bug-prone.

Annotation-induced (29/10.7%) refactoring engine bugs are the third most in our studied bug reports with input program. Annotation is a frequently used feature in Java, and previous research [106] has discovered that it can lead to bugs in program analysis and optimization tools. By analyzing our studied bug reports together with the discussions and fixes, we summarize three factors that affect the parser, interpreter, and code modification of refactoring tools: (1) Incomplete semantics. Refactoring aims to alter the structure of the code without changing its semantics. However, annotations can carry semantic meaning that affects program behavior, and refactoring tools may not understand or respect these semantics to avoid altering program behavior. Figure 32[55] gives an example where the refactoring of the push-down method produces a compilation error for the input program in Listing 47 because @Override is not removed. @Override represents the relationship among classes in Java that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes, and refactoring engine developers ignore it when rewriting the AST for program transformation, resulting in a compilation error as the @Override annotation is mistakenly retained in the refactored program in Listing 48. (2) Complexity of annotation processing. Some runtime annotations (e.g., @Deprecated) may influence the behavior of the program when it is running,

[54] https://bugs.eclipse.org/bugs/show_bug.cgi?id=393829
[55] https://bugs.eclipse.org/bugs/show_bug.cgi?id=394728

```
1  class PushDownBug1 {
2      static abstract class C {
3  -       abstract void m();
4      }
5      static final C F = new C() {
6          @Override
7          void m() {
8          }
9      };
10 }
```

Listing 47. Code before refactoring.

```
1  class PushDownBug1 {
2      static abstract class C {
3      }
4      static final C F = new C() {
5          @Override
6          void m() {
7          }
8      };
9  }
```

Listing 48. Code after refactoring.

Fig. 32. Eclipse-394728: "Push Down" refactoring does not remove @Override leading to compilation error in Eclipse 4.2.1.

which is typically outside the direct manipulation scope of static refactoring tools. (3) Complex transformations for annotations. Annotations are often used in contexts that involve reflection or dynamic code transformations, which are inherently complex to refactor. Refactoring tools must analyze and modify such annotations intelligently, a process that is error prone due to the dynamic and interconnected nature of these frameworks.

Finding 10: Annotation-induced (29/10.7%) refactoring engine bugs are the third most in our studied bug reports with input program. The reasons include: (1) Lack of consideration of the semantics of annotations, (2) Complexity of annotation processing, which may be processed at various phrases: compile-time (e.g., @Override), runtime (e.g., @Deprecated), and deployment (e.g. @WebServlet in Java Servlet API), (3) Complex transformations for annotations.

There are some (18/6.7%) bugs related to code comments. For example, not updating the corresponding code comments after renaming the method parameters, accidentally deleting the code comments when extracting local variables, and refactoring tools might deny pulling up a variable as a field just because there is a code comment following the variable. After analyzing the attached patches, we identified the following reasons: (1) Comment-code dependency: comments sometimes rely on specific code constructs, such as method names, variable names, or parameter names, to maintain their relevance and accuracy. When refactoring tools modify these code elements, they may inadvertently render associated comments outdated or incorrect. For example, renaming a method parameter without updating the accompanying comment describing its purpose can lead to confusion. (2) Semantic gaps: Comments often provide additional context or explanations about the code, but they are not formally part of the code's syntax or semantics. Refactoring engines primarily operate on the code's structure and semantics defined by the programming language, making it challenging to bridge the gap between code changes and their corresponding comments. For example, when Eclipse performs "Move Class to New File" refactoring, some comments are deleted in both the new file and the old file [91]. Listing 50 in Figure 33 shows an example in which a code comment is incorrectly deleted by Eclipse 3.6 when performing inline local variable refactoring for the input program in Listing 49. As the Eclipse developer stated: "the problem comes from the fact that this segment is deleted by the inlining operation without taking care of the attached comment"[56]. An existing study [69] tried to retain comments while refactoring code, but it is still limited in supporting multi-languages and cannot handle complex situations (e.g., code and comment tracing [84, 94], parsing issues for input programs with comments [90], and considering various comment positions [92]).

[56] https://bugs.eclipse.org/bugs/show_bug.cgi?id=295200

```
1 public void testIt() {
2 -  // blah
3 -  String test = "";
4 -  System.out.println(test);
5 }
```

Listing 49. Code before refactoring.

```
1 public void testIt() {
2 +  System.out.println("");
3 }
```

Listing 50. Code after refactoring.

Fig. 33. Eclipse-295200: comment is wrongly deleted when performing inline local variable refactoring in Eclipse 3.6.

Table 8. Chi-squared test of independence for each category pair.

Category pair	χ^2	p-value	Description
(Root Cause, Symptom)	328.638	1.884e−34	Related
(Refactoring type, Root cause)	714.225	0.003	Related
(Refactoring type, Symptom)	1405.076	1.000	Not related
(Input program characteristic, Refactoring type)	4374.049	1.941e−6	Related
(Input program characteristic, Root cause)	377.659	1.198e−4	Related
(Input program characteristic, Symptom)	935.347	1.867e−22	Related

Finding 11: Refactoring involving comments interleaved with code might cause refactoring to fail, this contributes to 18/6.7% of our studied bugs. After analyzing the bug reports and patches, we conclude the reasons as: (1) Comment-code dependency, and (2) Semantic gaps between code and comments. This finding calls for further studies on comment-aware auto-refactoring tools.

4.5 RQ5: Statistical Analysis

Following prior studies [14, 18, 24, 59, 106], we first start by analyzing the relationship between the root cause of the bug and the symptom, as their relationship can help to understand the impact of various root causes on refactoring engines. To validate if two categories (e.g., symptom and root cause) are related, we first performed a Chi-squared test [43] of independence. Then, to see which pairs of values have statistically significant relationships, for example, to confirm whether "Incorrect Transformation" and "Compile Error" (as values) are really related, we performed a pairwise post hoc analysis using Bonferroni Adjustment [58]. We conducted the same analysis for the rest of category pairs: (Refactoring type, Root cause), (Refactoring type, Symptom), (Input program characteristic, Refactoring type), (Input program characteristic, Root cause), and (Input program characteristic, Symptom).

The statistics of the Chi-squared test for each pair of categories is shown in Table 8. Except for (Refactoring type, Root cause), the results for the other five pairs are significant (p-value < 0.05). This indicates that there are correlations between the variables of those pairs. For root cause and symptom, the Chi-squared statistic is 328.638, the p-value is 1.884e−34 (< 0.05), which means that these two categories are strongly correlated. Post hoc analysis shows that "Failed selection parsing" and "Refactoring not available" (adjusted p-value = 1.374e−34), "Incorrect transformations" and "Comment related" (adjusted p-value = 0.008), "Others" and "Bad performance" (adjusted p-value = 2.334e−17), "Incorrect type resolving" and "Failed refactoring" (adjusted p-value = 0.019) have statistically significant associations. Figure 34[57] shows an example in which "Failed selection parsing" results in "Refactoring not available". In this example, refactoring is not available for the variable "xy" on line 4 for the input program in Listing 51. This happens because

[57] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/156

```
@SuppressWarnings("preview")public class X {
public static void printLowerRight(Rectangle r) {
        int res = switch(r) {
           case Rectangle(ColoredPoint(Point(int xy, int y), Color c), ColoredPoint lr) r1  -> {
             ...
        };
    }
    ...
}
record Point(int x, int y) {}
enum Color {RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

Listing 51. Input program.

Fig. 34. Eclipse-156: Refactor does not work for record pattern.

```
Assert.isTrue(fCUnit.isWorkingCopy(), fCUnit.
      toString());
```

Listing 52. Code before optimization.

```
Assert.isTrue(fCUnit.isWorkingCopy(), !fCUnit.
      isWorkingCopy()?fCUnit.toString():"");
```

Listing 53. Code after optimization.

Fig. 35. Eclipse-570587: unnecessary loss of performance in refactoring.

some newly introduced programming language features (e.g., Java Record pattern[58] in Figure 34) in the newest version of JDK have not been fully supported in IDEs, and the selected program elements cannot be parsed for the refactoring operation, thus resulting in "Refactoring not available". As the engine developer stated in the bug report — "Currently, there is no AST node for variables such as xy and lr. This should be revisited after Record Pattern DOM AST changes are merged". Figure 35[59] lists an example to illustrate the correlation between "Others" and "Poor performance". The method toString() in Listing 52 is called in the assertion in the implementation of refactoring engine, however, as the engine developer stated — "the performance penalty for this use case is higher, ... we should change the code to only call toString() if really needed". Thus, a time-consuming call causes "Poor performance". As shown in Listing 53, engine developers optimize it by adding a conditional branch, so the time-consuming call is only called when needed.

The Chi-squared statistic for (Refactoring type and Root cause) is 714.225, the p-value is 0.003 (< 0.05), indicating that two categories are strongly correlated. Post hoc analysis shows that various pairs of related variables, for example "Use supertype wherever possible" and "Incorrect Type Resolving" (adjusted p-value = 5.095e−4), applying super-type whenever possible refactoring involves type resolving, so it is reasonable to have correlation with "Incorrect Type Resolving". "extract method" and "incorrect flow analysis" (adjusted p-value = 0.002) are also correlated because flow analysis is required when refactoring of the extraction method is performed. For the (Input program characteristic, Refactoring type), the Chi-squared statistic is 4374.049 and the p-value is 1.941e−6, which means certain input program characteristics could cause some refactoring types to fail. We could leverage this finding to design error-prone input programs for certain refactoring types. Post hoc analysis shows that few pairs of variables are related. For example, "inner class" and "Move Member Type to New File" (adjusted p-value = 9.374e−9), this means "Move Member Type to New File" refactoring is error prone for inner class in the input program. "Comment related" and "change method signature" (adjusted p-value = 2.865e−4) indicates that when performing refactoring of the "change method signature", the comment related to the method may not be correctly updated. The characteristic of the input program is also

[58] https://docs.oracle.com/en/java/javase/22/language/record-patterns.html
[59] https://bugs.eclipse.org/bugs/show_bug.cgi?id=570587

```
1 String[] parts = someString.split(",|<caret>\t");
```

Listing 54. A statement in the input program.

Fig. 36. IDEA-90322: cannot use extract variable inside string containing injected language.

correlated with the root cause (Chi-squared statistic is 377.659 and the p-value is 1.198e−4). For example, "var type" and "Incorrect Type Resolving" (adjusted p-value = 0.002) are related because refactoring involving var type[60] in Java requires type resolving. "special string" and "failed selection parsing" (adjusted p-value = 0.001) are related because some special strings cannot be correctly parsed by IDEs. For example, as shown in Figure 36[61], Listing 54 gives an example in which IntelliJ IDEA cannot parse a string containing an injected language. When moving the cursor to the position specified by "<caret>", it is expected to offer to extract the string into a variable; however, IntelliJ IDEA selects the whole line and complains that it is not an expression. (Input program characteristic, Symptom) is also correlated (Chi-squared statistic is 935.347 and the p-value is 1.867e−22). For example, toString() and poor performance (adjusted p-value = 7.973e−55) as input program containing the toString() method is time-consuming, resulting in poor performance. Input programs containing comments are likely to result in comments-related symptoms (adjusted p-value = 1.959e−42), and refactoring for input programs having a local class might result in compiler error (adjusted p-value = 1.138e−12). We put the complete statistical analysis result for all category pairs in Table 8 in our supplementary material[62].

Finding 12: Through our statistical analysis, we conclude that (Root Cause, Symptom), (Refactoring type, Root cause), (Input program characteristic, Refactoring type), (Input program characteristic, Root cause) and (Input program characteristic, Symptom) are correlated, while (Refactoring type, Symptom) is not correlated.

5 Transferability study

5.1 Bug transferability study cross refactoring engines

In this section, we conduct a transferability study using the historical refactoring engine bugs. Our hypothesis is that *refactoring engines like Eclipse, IntelliJ IDEA, and Netbeans all support the same frequently used refactoring types. Thus, for common refactoring, bugs in one refactoring engine might also exist in other refactoring engines.* Therefore, we could potentially uncover bugs by cross-validating historical bug reports for each refactoring engine. Specifically, the procedures for our transferability study are as follows.

(1) We first identify the refactoring types that are supported in the three refactoring engines studied: Eclipse, IntelliJ IDEA, and Netbeans. We refer to these as Common Refactoring Types (CRT). From this set, we select the top 10 most error-prone refactoring types based on our Finding 1. The decision to focus on the top 10 is motivated by their higher likelihood of triggering bugs, as well as the fact that manually validating all CRTs would be nontrivial and time-consuming.
(2) Bug reports related to the top 10 CRTs are extracted from the RefactorBench dataset and filtered for cross-validation. These bug reports are categorized into three partitions which correspond to bug reports from Eclipse, IntelliJ IDEA, and Netbeans, respectively. For each partition, we cross-validate the bug reports against the

[60] https://docs.oracle.com/en/java/javase/17/language/local-variable-type-inference.html
[61] https://youtrack.jetbrains.com/issue/IDEA-90322
[62] https://github.com/HaiboWang1992/RETester/tree/main/statistic_analysis

Table 9. Bugs found by our transferability study distributed by their sources.

Source \ Target	Eclipse	IntelliJ IDEA	Netbeans	**Total**
Eclipse	-	24	34	58
IntelliJ IDEA	12	-	47	59
Netbeans	1	12	-	13
Total	13	36	81	130

other two target refactoring engines. For example, bug reports from Eclipse are validated in both IntelliJ IDEA and Netbeans by analogy.

(3) We obtain the latest versions of Eclipse (2024-03), IntelliJ IDEA (2024.1.2), and Netbeans (Apache NetBeans IDE 22) at the time of our study for testing. For each bug report within a partition, we first check the bug tracking systems of the target IDEs to avoid duplicate reports before manual validation. The search process involves looking for refactoring-type keywords and symptoms. If any bug reports with *similar* input programs are found, we consider the bug already reported and exclude it from further validation. Specifically, we check if the input programs are *similar* by checking whether the two input programs have the same set of program elements (e.g., same class declaration, class reference, class inheritance relationship, field declaration, field reference, method declaration, method logic), allowing only for differences in the name of the code elements (e.g., class name, filed name, method name, variable name).

(4) If no duplicated bug has been reported, we proceed with manual reproduction using the provided input program, and refactoring procedures. If any of the previously mentioned symptoms in RQ2 occurs, it is classified as a bug in the target refactoring engine.

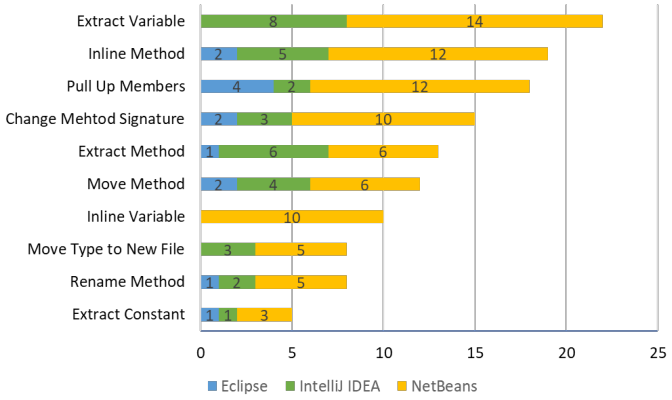

Fig. 37. Bugs found by our transferability study across each target IDE.

Figure 37 presents the number of bugs identified in each target IDE during our transferability study, focusing on the top 10 types of refactoring. For each target IDE, the reported bugs are newly discovered and had not been previously reported. In total, 130 bugs were discovered in our study [78], including 13 in Eclipse, 36 in IntelliJ IDEA, and 81 in Netbeans. Among all refactoring types, "Extract Variable" triggered the most bugs (22), followed by "Inline Method" (19 bugs) and "Pull Up Members" (18 bugs). Netbeans contributed the largest share, with 83 new bugs — accounting

Table 10. Submitted bugs revealed by our transferability study.

ID	Source	Target	Issue No.	Title	Symptom	Status
I-1	Eclipse	IDEA	354122	Extract Method refactoring produces refactored program contains syntax error	Compile error	Fixed
I-2	Eclipse	IDEA	354116	Refactoring erroneously qualifies calls inside the anonymous inheritor of the outer class	Behavior change	Fixed
I-3	Eclipse	IDEA	354039	Refactoring changes the behavior of my program	Behavior change	Confirmed
I-4	Eclipse	IDEA	354040	Change Signature refactoring fail to perform the change as shown in the preview	Fail to refactoring	Confirmed
I-5	Eclipse	IDEA	355272	Pull Members Up refactoring for the classes using generic types producing uncompilable program	Compile error	Submitted
I-6	Eclipse	IDEA	355271	Move Inner Class to Upper Level refactoring fails for inner class using generic type	Compile error	Submitted
I-7	Eclipse	IDEA	355273	Move inner class to different package refactoring result in syntax error	Compile error	Submitted
I-8	Eclipse	IDEA	355200	Move Inner Class to Upper Level refactoring produces uncompilable program for classes in different package	Compile error	Submitted
I-9	Eclipse	IDEA	354991	Pull method up and make method abstract result in refactored program contains syntax error	Compile error	Submitted
I-10	Eclipse	IDEA	355423	Ïntroduce Variable" refactoring changes the behavior of the program	Behavior change	Won't fix
I-11	Eclipse	IDEA	355421	Introduce Variable refactoring changes the semantic of the program because IDEA does not identify the statements that may change the value of the extracted expressions	Behavior change	Submitted
I-12	Eclipse	IDEA	354041	Extract method refactoring issue about the parameter of the extracted method	Behavior change	Won't fix
I-13	Eclipse	IDEA	354042	Type Migration refactoring produce refactored program contains syntax error	Compile error	Shelved
I-14	Eclipse	IDEA	353991	Fail to rename class	Fail to refactoring	Submitted
E-1	IDEA	Eclipse	1529	Inline the method which contains super keyword in a static method result in the refactored program has syntax error	Compile error	Fixed
E-2	IDEA	Eclipse	1530	Move static members refactoring results in behavior change	Behavior change	Fixed
E-3	IDEA	Eclipse	1531	Avoid the perform of rename refactoring at the implicit enum elements (e.g. values() method)	Compile error	Fixed
E-4	IDEA	Eclipse	1532	[Bug][Pull Up Refactoring] Pull up refactoring produces uncompilable program	Compile error	Fixed
E-5	IDEA	Eclipse	1533	Pull up method refactoring for method in the inner class fails	Compile error	Fixed
N-1	Eclipse	NetBeans	7428	Introducing method on ïnstanceof" caused compile error	Compile error	Submitted
N-2	Eclipse	NetBeans	7427	Fail to introduce method	Fail to refactoring	Submitted

The issues of IntelliJ IDEA, Eclipse, and Netbeans can be found at https://youtrack.jetbrains.com/issue/IDEA-XXX, https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/XXX, and https://github.com/apache/netbeans/issues/XXX, where "XXX" can be replaced with the concrete numbers in **Issue No.**. "Shelved" means the bug is confirmed but it is not included in the foreseen product plans.

for 63.8% of the total — indicating lower reliability in handling refactorings compared to the other IDEs. In contrast, Eclipse showed the fewest issues, with only 13 identified bugs, suggesting that it is the most robust among the three engines studied. Table 9 details the distribution of bugs according to their sources. The column "Source" denotes the IDE from which the historical bug reports originate, while the column "Target" indicates the IDEs where new bugs were triggered. Specifically, 58 historical bug reports from Eclipse revealed new bugs in other IDEs: 24 in IntelliJ IDEA and 34 in Netbeans. Similarly, historical reports from IntelliJ IDEA led to the discovery of 12 bugs in Eclipse and 47 in Netbeans. Lastly, 13 historical bug reports from Netbeans triggered 1 new bug in Eclipse and 12 in IntelliJ IDEA.

Table 10 lists the 22 bugs we submitted at the time of manuscript submission. We are currently preparing for the submission of the rest of the bugs. The column "Source" indicates the origin of the historical bug report, while the column "Target" column specifies the refactoring tool being tested. The columns "Issue No." and "Title" provide detailed information on each submitted issue, and the column "Symptom" summarizes the symptoms associated with each bug. The final column details the current status of each submitted issue. By the time of the submission of this paper, seven of the issues had been fixed and three had been confirmed.

Figure 38 illustrates the input program in Listing 55 and the refactored result in Listing 56 produced by IntelliJ IDEA 2024.1.2 for issue I-10 in Table 10. Before refactoring, the program output was: "hello:1, hello:2, hello:3, hello:4". However, the output of the refactored program became: "hello:1, hello:1, hello:1, hello:1." This occurred because each invocation of the "foo" method updates the state of "i", producing different outputs for each call. However, IntelliJ IDEA extracted "foo("hello")" as a variable and replaced each invocation with the extracted variable, disregarding the

```
1  public class A {
2      static String[] arr = {"1", "2", "3", "4"};
3      static int i = 0;
4      private static String foo(String str) {
5          return str + ":" + arr[i++];
6      }
7      public static void main(String[] args) {
8  -       System.out.println(foo("hello"));
9  -       System.out.println(foo("hello"));
10 -       System.out.println(foo("hello"));
11     }
12 }
```

Listing 55. Code before refactoring.

```
1  public class A {
2      static String[] arr = {"1", "2", "3", "4"};
3      static int i = 0;
4      private static String foo(String str) {
5          return str + ":" + arr[i++];
6      }
7      public static void main(String[] args) {
8  +       String hello = foo("hello");
9  +       System.out.println(hello);
10 +       System.out.println(hello);
11 +       System.out.println(hello);
12     }
13 }
```

Listing 56. Code after refactoring.

Fig. 38. IDEA-355423: "Introduce Variable" refactoring changes the behavior of the input program in IntelliJ IDEA 2024.1.2.

side effects of the extracted expression. As a result, the output was incorrect: "hello:1, hello:1, hello:1, hello:1", since the "foo" method is only invoked once in the refactored program. The original bug was reported in Eclipse [87], and the developers addressed the issue by implementing side effect analysis during variable extraction. Although we identified the same issue in the latest version of IntelliJ IDEA and reported it to its developers, they classified it as a usability problem and were unwilling to fix it by explaining that "We don't guarantee that there will be no semantics change when 'all occurrences' is selected, and it is the user's responsibility to decide whether it is safe or not"[63]. A similar issue was observed in I-12 in Table 6. This suggests that while different IDEs support the same refactoring types, their handling of complex operations, such as side-effect analysis, can vary slightly. However, during our interactions with the developers of the refactoring engine, they largely agreed with our reported bugs, resulting in seven fixes and three confirmations. One issue (I-13) was confirmed by the IntelliJ IDEA developers but was not included in their immediate product plans, leading them to assign it the status of "Shelved".

5.2 Bug regression study for each refactoring engine

Following our study in Section 5.1, we also conducted an experiment to reproduce the bugs within the same engine that caused the bug reports. The purpose of this study is to ensure that the latest version of that engine correctly prevents the issue from raising again. We aim to find regression bugs [98], which are claimed to have been fixed by the engine developers but still work incorrectly in the latest version. Specifically, we first filter the bug reports that belong to the top 10 most error-prone refactoring types based on our Finding 1, as in Section 5.1. Then, we classify these bugs according to refactoring engines, resulting in 132 historical bug reports from Eclipse, 120 historical bug reports from IntelliJ IDEA, and 22 historical bug reports from Netbeans, respectively. Based on the input programs and the reproduce procedures in the bug reports, we manually reproduced each historical bug in its corresponding latest version refactoring engine at the time of our study, including Eclipse (2024-03), IntelliJ IDEA (2024.1.2), and Netbeans (Apache NetBeans IDE 22). If the current bug is reproducible, we take it as a regression bug.

As shown in Table 11, we found four regression bugs in total, including one bug from Eclipse and three bugs from IntelliJ IDEA. Most of the bugs cannot be reproduced in the latest version of refactoring engines because developers have added the corresponding input programs in bug reports into test suite, thus developers could repair any new regression bugs by running the regression test suite after each update. For the regression bugs in Table 11, we analyzed their patches and found out that developers tend to create their own tests rather than add the input program in bug

[63] https://youtrack.jetbrains.com/issue/IDEA-355423

Table 11. Regression bugs revealed through our transferability study.

ID	URL	IDE	Version	Symptom
RB-1	https://bugs.eclipse.org/bugs/show_bug.cgi?id=111056	Eclipse	2024-09	Compile error
RB-2	https://youtrack.jetbrains.com/issue/IDEA-104286	IDEA	2024.1.2	Compile error
RB-3	https://youtrack.jetbrains.com/issue/IDEA-144043	IDEA	2024.1.2	Compile error
RB-4	https://youtrack.jetbrains.com/issue/IDEA-118166	IDEA	2024.1.2	Failed refactoring

```
1 public class A {}
2 public class B extends A {
3   public void m() {} // pull up and make abstract
4   void n() {
5     new A();
6   }
7 }
```

Listing 57. Input program in bug report.

```
1 class A {}
2 class B extends A {
3    public void test(){}
4 }
```

Listing 58. The input program added in test suite.

Fig. 39. RB-3: input program in bug report and the input program added in test suite.

reports. For example, for RB-3 in Table 11, the input program is shown in the Listing 57 in Figure 39, and the input program added in the test suite[64] is in Listing 58. The developers omit the method n() in line 4 of Listing 57, however, it is the critical code for this bug. In this bug, method m() is pulled up to class A and made abstract, thus class A is changed to abstract silently because it contains abstract method m(). Method n() initiates class A in line 5, this could result in compile error after class A is changed to abstract. However, the input program in Listing 58 that IntelliJ IDEA developers added in test suite ignores method n(), which is the critical code to trigger bug. In this case, even though the developers of IntelliJ IDEA have changed the status of RB-3 to fixed, it has not been fixed completely. We have reported this bug to the developers of IntelliJ IDEA, and they have confirmed it[65].

5.3 The performance of the existing refactoring engine testing tools to identify transferred bugs

In this chapter, we study the performance of the existing refactoring engine testing tools to identify our transferred bug. Specifically, we focus on the following tools:

SAFEREFACTOR. SAFEREFACTOR [65] is a test-based refactoring validation tool to detect behavioral changes in Java programs. The tool generates test suites that are useful to pinpoint non-behavior-preserving transformations. As shown in Figure 40, the whole process has the following steps:

(1) The developer selects the refactoring to be applied in the input program and uses SAFEREFACTOR. The plugin uses the Eclipse refactoring API to generate a target program based on the desired transformation. A static analysis automatically identifies methods that are common (same signature) in both source and target versions.

(2) Step 2 aims at generating unit tests, using Randoop [53], for the common methods identified in Step 1. The same tests can be run on the source and target versions, since it generates tests only for the common methods.

(3) In Step 3, the plug-in runs the generated test suite on the source.

(4) Next, it runs the same test suite in the target version.

[64] https://github.com/JetBrains/intellij-community/commit/9ce5f869de988ee4e164f9f54632ba11c9806e9a
[65] https://youtrack.jetbrains.com/issue/IDEA-354991

Table 12. The effectiveness of SAFEREFACTOR and ASTGEN to detect the bugs revealed by our transferability study.

ID	Source	Target	Issue No.	Refactoring	Symptom	SAFEREFACTOR	ASTGEN
I-1	Eclipse	IDEA	354122	Extract Method	Compile error	Y	NA
I-2	Eclipse	IDEA	354116	Make Static	Behavior change	N	NA
I-3	Eclipse	IDEA	354039	Introduce Parameter	Behavior change	N	N
I-4	Eclipse	IDEA	354040	Change Signature	Failed refactoring	N	N
I-5	Eclipse	IDEA	355272	Pull Members Up	Compile error	Y	N
I-6	Eclipse	IDEA	355271	Move Inner Class	Compile error	Y	N
I-7	Eclipse	IDEA	355273	Move Inner Class	Compile error	Y	N
I-8	Eclipse	IDEA	355200	Move Inner Class	Compile error	Y	N
I-9	Eclipse	IDEA	354991	Pull Method Up	Compile error	Y	N
I-10	Eclipse	IDEA	355423	Introduce Variable	Behavior change	N	NA
I-11	Eclipse	IDEA	355421	Introduce Variable	Behavior change	N	NA
I-12	Eclipse	IDEA	354041	Extract Method	Behavior change	N	NA
I-13	Eclipse	IDEA	354042	Type Migration	Compile error	Y	NA
I-14	Eclipse	IDEA	353991	Rename Class	Failed refactoring	N	N
E-1	IDEA	Eclipse	1529	Inline Method	Compile error	Y	NA
E-2	IDEA	Eclipse	1530	Move Static Memebers	Behavior change	N	NA
E-3	IDEA	Eclipse	1531	Rename Method	Compile error	Y	N
E-4	IDEA	Eclipse	1532	Pull Up	Compile error	Y	N
E-5	IDEA	Eclipse	1533	Pull Up	Compile error	Y	N
N-1	Eclipse	NetBeans	7428	Introduce Method	Compile error	Y	NA
N-2	Eclipse	NetBeans	7427	Introduce Method	Failed refactoring	N	NA

Y: Current bug can be detected; **N**: Current bug cannot be detected; **NA**: Since ASTGen is a template-based test generation tool, it relies on predefined templates to generate input programs for testing refactoring engines. Currently, ASTGen only provides 24 predefined templates for eight refactoring types. NA indicates that the refactoring is out of its predefined templates scope.

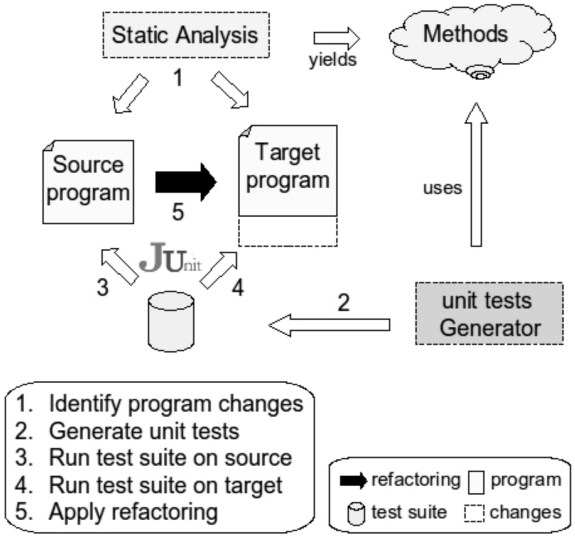

Fig. 40. The workflow of SAFEREFACTOR [65].

(5) If a test passes in one of the versions and fails in the other, the plugin detects a behavioral change and reports to the user. Otherwise, the programmer can have more confidence that the transformation does not introduce behavioral changes. It can also report compilation errors that may be introduced by refactoring tools.

SAFEREFACTOR could also serve as an out-of-box tool, it requires the program before and after refactoring as input. During our experiment, for each bug report, we extract the input program before and after refactoring as

```
1  public class A<T> {
2-     class Inner {
3-         T foo;
4-     }
5      T foo() {
6          Inner inner = new Inner();
7-         return inner.foo;
8      }
9  }
```

Listing 59. Code before refactoring.

```
1  public class A<T> {
2      T foo() {
3          Inner inner = new Inner();
4+         return inner.foo;
5      }
6  +}
7  +class Inner {
8  +    T foo;//syntax error
9  }
```

Listing 60. Code after refactoring.

Fig. 41. IDEA-355271: Move Inner Class to Upper Level refactoring fails for inner class using generic type in IntelliJ IDEA 2024.1.2.

input. The experimental result is shown in Table 12. The bugs that have "compile error" symptom can all be detected by SAFEREFACTOR. This is because if the refactored program cannot be compiled while the input program before refactoring could, it is very easy to identify according to this symptom. Three bugs that have the symptoms of "Failed refactoring" (I-4, I-14, and N-2) cannot be discovered by SAFEREFACTOR. For I-4[66] and I-14[67], the input program before and after the refactoring of the change signature is the same, so SAFEREFACTOR cannot identify it as a bug. For N-2[68], the refactoring engine exhibits a warning message to stop the refactoring, SAFEREFACTOR cannot obtain the refactored program as input. It is worth mentioning that although SAFEREFACTOR is a test-based refactoring validation tool, it does not discover all "Behavior change" bugs (I-2, I-3, I-10, I-11, I-12 and E-2) in Table 12. After we analyzed the input programs for those bugs, we concluded two reasons: (1) SAFEREFACTOR generates tests only for the common methods before and after refactoring. (2) According to an existing study [63], traditional test generation tools (i.e., Randoop [53] and EvoSuite [17]) are not efficient for refactoring validation.

ASTGen. ASTGen [10] is a framework to generate structurally complex input programs for the refactoring engine. It allows developers to write templates whose executions produce input programs with structural properties that are relevant to the refactoring being tested. For example, to test the Rename Field refactoring, the input program should have a class that declares fields and variables with potential name clashes. ASTGen has few drawbacks. First, ASTGen requires developers to manually write predefined templates to generate input programs. However, developers may not have sufficient knowledge or insight about the types of program structure that are more likely to trigger bugs. Moreover, modern refactoring engines in IDEs like IntelliJ IDEA support a variety of refactoring types, which makes the manual design of templates labor intensive and impractical. Finally, for each refactoring, the input programs that are error prone could be diverse and complex, thus depending on manually designing templates makes the tool less effective and scalable. Currently, ASTGen provides only 23 predefined templates for eight refactoring types, as shown in Table 13. The other refactorings are out of scope (marked NA in Table 12).

For the input program under refactoring in each bug report, we manually compare it against the predefined templates (T1 – T23 in Table 13) provided by ASTGen, we check if the templates define the necessary structures to generate the input program, including field declaration, field reference, method declaration, method reference, class declaration, class relationship. If the structure is not defined in the templates, ASTGen cannot generate the target input program, thus cannot trigger the bug in the refactoring engine. As shown in Table 12, ASTGen cannot discover any bug because it provides a very limited number (i.e., 23) of predefined templates to generate input programs. For example, I-6[69] in Figure 41 gives an example where IntelliJ IDEA produces a refactored program (Listing 60) that has a syntax error

[66] https://youtrack.jetbrains.com/issue/IDEA-354040
[67] https://youtrack.jetbrains.com/issue/IDEA-353991
[68] https://github.com/apache/netbeans/issues/7427
[69] https://youtrack.jetbrains.com/issue/IDEA-355271

Table 13. Refactoring types together with the predefined templates that ASTGen covers.

Refactoring	Granularity	NO.	Template Structure	Template Description
Rename	Class	T1	ClassRelationships	Class A referenced from class B, located in various places in relation to A. Refactor to rename A.
	Method	T2	SingleClassMethodReference	A single class A containing a method m referenced from a method mPrime. mPrime may override m. Refactor to rename m to n.
	Field	T3	SingleClassFieldReference	A single class A with field referenced in some expression. Refactor to rename f.
		T4	DualClassFieldReference	Two classes A and B. A declares a field and B references it in some expression. Refactor to rename f.
Encapsulate	Field	T5	ClassWithArrayField	Encapsulate a field in Class A where the field has various array types. Refactor to encapsulate f.
		T6	DualClass2FieldsParentDeclaration	Two classes A and B. Two optionally referenced fields. A always includes the declaration of the first field and an expression that referenec this field. Refactor to encapsulate f.
		T7	DualClassFieldGetterSetter	Two classes A and B. There is one field declared in class A and a getter/setter declared in one of these classes.
		T8	DualClassFieldReference	Two classes A and B. A declares a field and B references it in some expression. Refactor to encapsulate f.
		T9	SingleClass2FieldsOptInitialized	A single class A with two fields optionally referenced. Refactor to encapsulate f.
		T10	SingleClassFieldReference	A single class A with field referenced in some expression. Refactor to encapsulate f.
Push Down	Field	T11	DualClass2FieldsParentDeclaration	Two classes A and B. Two optionally initialized fields. A always includes the declaration of the first field.
		T12	DualClassFieldReference	Two classes A and B. A declares a field and B references it in some expression. B extends A. Refactor to push f down to B.
	Method	T13	DualClassMethodParent	Two classes A and B. A declares a method and B references it in some expression. B extends A. Refactor to push m down to B.
Pull Up	Field	T14	DualClass2FieldsChildDeclaration	Two classes A and B. Two optionally initialized fields. A always includes the declaration of the first field.
		T15	DualClassFieldReference	Two classes A and B. A declares a field and B references it in some expression. B extends A. Refactor to push f down to B.
		T16	TripleClass2FieldsChildDeclaration	Three classes A, B and C. Two optionally initialized fields. A always includes the declaration of the first field and B extends C
	Method	T17	DualClassMethodChild	Two classes A and B. B declares one method and another one that references it in some expression. B extends A. Refactor to pull m up to A.
		T18	TripleClassMethodChild	Three classes A, B and C. B declares method m and another one that references it in some expression. B extends C. Refactor to pull m up to A.
Change Signature	Change Return Type	T19	SingleClassMethodReference	A single class containing a method m referenced from mPrime. mPrime may overload m. Refactor to change the return type of m
	Remove Parameter	T20	MethodWithParameterReference	A single class A with a method m. m has an argument a that is referenced in the method in various ways. Refactor by removing parameter.
		T21	SingleClassMethodReference	A single class A with a method m. m has a parameter that is not referenced in the method but is referenced from another method mPrime. Refactor by removing the parameter.
Member To Top	Inner Class	T22	ClassRelationships	Two classes, A and B. B is an inner class of A. A and B can both reference each other.
		T23	DualClassFieldReference	Two classes A and B. B is inner to A. A declares a field and B references it in some way. Refactor by moving B to the top level

after moving an inner class to the upper level. The critical input program characteristics to trigger this bug are the generic type of the class as shown in Listing 59. However, ASTGen does not provide any template to generate this kind of input program for "Member To Top" refactoring (i.e., T22 and T23 in Table 13), and thus cannot trigger this bug.

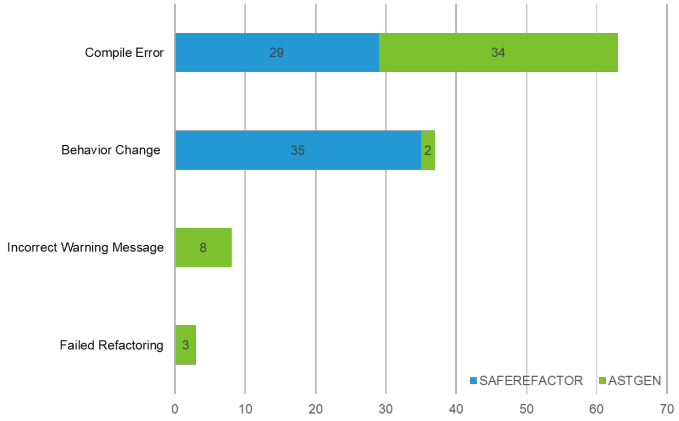

Fig. 42. Distribution of bug symptoms for the bugs reported by ASTGen and SAFEREFACTOR.

To further analyze what kind of bugs ASTGen and SAFEREFACTOR can detect beyond our transferred bugs, we collected all reported bugs listed on their official websites[70][71]. Specifically, we focus on the bug reports within our studied refactoring engines (i.e., Eclipse, IntelliJ IDEA, and Netbeans) and bug reports marked as "Not Answered" and "Not a bug" are excluded, leading to 47 and 64 bug reports for ASTGen and SAFEREFACTOR, respectively. In addition, we classified those bug reports according to their bug symptoms. Figure 42 presents the distribution of bug symptoms for bugs that we collected across different tools. As shown in Figure 42, SAFEREFACTOR could detect two types of bugs. Specifically, among the 64 bugs reported by SAFEREFACTOR, 29 bugs cause "Compile Error" and 35 bugs induce "Behavior Change". Meanwhile, ASTGen could detect four types of bugs. In addition to the bugs showing the symptoms of "Compile Error" (34) and "Behavior Change" (2), it can also detect "Incorrect Warning Message" (8) and "Failed Refactoring" (3) bugs. This could be attributed to the various oracles designed in ASTGen (e.g., warning status oracle and differential oracle) compared to relying solely on the passed / failed tests generated by SAFEREFACTOR. However, their reported bugs only cover four out of nine symptoms as we found in RQ2 (Section 4.2). Those four symptoms take up to 68.53% (355 out of 518) of our studied historical bugs, the other 31.46% bugs are possibly beyond their ability. For example, SAFEREFACTOR cannot detect code comment-related bugs since it can only focus on the functional behavior of the program during test generation by Randoop [53]. The same situation also happens in ASTGen, as it only designs the program generators (e.g., field reference generator and class relationship generator), and code comment generator is not considered at all in their tool. In addition, the detection for the bug types they claimed to be supported could be in a low effectiveness. For example, although SAFEREFACTOR targets behavior-changing bugs, in Table 12, none of the five behavior-changing bugs (i.e., I-2, I-3, I-10, I-11, and I-12) can be detected by it. This is attributed to the design of generating tests only for the common code before and after refactoring. Crash is the second highest bug symptom taking up 20.46% (106 out of 518) of our studied bugs. However, neither of ASTGen and SAFEREFACTOR reported any crash bug. For SAFEREFACTOR, its precondition to apply is that the refactoring engine should work, because it takes an original program and a refactored program as input. As for ASTGen, even though it includes "Does Crash Oracle", the effectiveness of the detection relies heavily on the template design. Currently, it only provides 23 predefined

[70] https://mir.cs.illinois.edu/astgen/
[71] http://www.dsc.ufcg.edu.br/~spg/saferefactor/experiments.htm

```java
public class A {
    public class BaseInner {}
    public class Outer {
        public int x = 0;
        public void foo(){};
        public class Inner extends BaseInner {
            void innerMethod() {
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

Listing 61. Input program.

Fig. 43. Input program extracted from a historical bug report.

Table 14. The prompt template used to perform mutations.

I will give the definition of the current refactoring, you need to understand it. You need to make sure the original refactoring could still be applied on the variant. **1**. **{Refactoring Type}**: **{Definition}** **2**. To expose more bugs in the refactoring engines, please generate edge case variant considering the **{Characteristic}** in current refactoring scenario. You need to generate the variant based on the input program, it is **{Input Program}**. **3**. You should give me the variant, the program elements to be refactored, and the procedures to refactoring. **4**. The generated variant should not contain any syntax errors. The Java program you generated should conformance with the JDK **{Version}** standard. Please generate one edge case variant considering different edge usage scenarios of **{Characteristic}** based on the input program.

templates for eight refactoring types, as shown in Table 13, and these templates are obviously not diverse enough for other refactoring types and to trigger multiple types of bugs according to their experimental results.

6 Guidelines and Implications

6.1 Guidelines

6.1.1 How our findings could inspire new testing method.

Actionable Suggestion 1. Design new refactoring engine testing techniques considering the bug triggering capability of the input program.

In Section 4.4 RQ4, we focus on the characteristics of the bug-triggering input program. We compile a taxonomy for these characteristics in Table 7 and distill five main findings (i.e., finding 8a, finding 8b, finding 9, finding 10, and finding 11). The revealed findings could be used to design new testing techniques to uncover refactoring engine bugs.

In this section, we conduct a preliminary experiment to demonstrate how these findings could be used in practice. Findings 8a and 8b show that input programs that have some Java language features (e.g., Java lambda expression) are more likely to trigger bugs in refactoring engines. Finding 10 concludes that input programs having complex class relationships (e.g., Java anonymous class) are more likely to result in refactoring engine failure. Based on these three findings, we designed a proof-of-concept mutation-based input program generator using LLM. First, we manually

```java
public class A {
    public class BaseTargetClass {}
    public class OriginalClass {
        public int data = 60;
        public void memberMethod() {}
        public class NestedOriginalClass extends
              BaseTargetClass {
            void setup() {
                new BaseTargetClass() {
                    // Pull this method up to
                         BaseTargetClass
                    void methodToBePulledUp() {
                        methodHelper();
                    }
                    void methodHelper() {
                        System.out.println("Helper
                              Method in Anonymous
                             Class: " + data);
                    }
                };
            }
        }
    }
}
```

Listing 62. The generated variant by applying the Java anonymous class transformation.

```java
public class A {
    public class BaseTargetClass {
            void methodToBePulledUp() {
                methodHelper();// syntax error
            }}
    public class OriginalClass {
        public int data = 60;
        public void memberMethod() {}
        public class NestedOriginalClass extends
              BaseTargetClass {
            void setup() {
                new BaseTargetClass() {
                    void methodHelper() {
                        System.out.println("Helper
                              Method in Anonymous
                             Class: " + data);
                    }
                };
            }
        }
    }
}
```

Listing 63. Apply pull up method refactoring for the variant produces uncompilable program in Eclipse (2024-09).

Fig. 44. One variant generated by LLM by applying the Java anonymous class transformation successfully uncover a new bug in Eclipse (2024-09).

extracted the input program from a historical bug report[72] from Eclipse. Listing 61 in Figure 43 shows the extracted input program. Then, we constructed prompts to instruct LLM to perform the mutation considering the characteristics revealed in our findings. The mutation prompt template is shown in Table 14. Specifically, we construct the prompt in a chain-of-thought [83] way. First, we ask LLM to understand the current refactoring by giving its refactoring type and its definition. The type of refactoring is obtained from the bug report; in this case, it is the "Pull Up Method". The corresponding definition is obtained from Martin Fowler's refactoring book [16]. Then, we ask LLM to perform the mutation based on the input program while considering the bug-triggering characteristics. We select the two most error-prone characteristics according to findings 8a, 8b and 9, which are the lambda expression (i.e., T1.1 in Table 7) and the anonymous class (i.e., T2.2 in Table 7). Then, we request LLM to provide the refactoring information for the generated variants including the program elements to be refactored and the refactoring procedures. Finally, we add some extra requirements like no syntax error and conformance with a specific JDK version (JDK 22.0.1 in this case). We use chatgpt-4o-mini as the underline LLM and invoke it using the OpenAI API[73]. For each characteristic, we ask the LLM to generate ten variants. The generated variants are then manually tested on the latest version of Eclipse (2024-09) and IntelliJ IDEA (2024.2.4) at the time of our study according to the refactoring information provided by LLM.

The generated variants successfully uncover three new bugs in Eclipse (2024-09) and IntelliJ IDEA (2024.2.4). We have reported those three new bugs to developers of refactoring engines, among which one[74] has been confirmed by IntelliJ IDEA developers, and two[75][76] have been fixed by Eclipse developers. Listing 62 in Figure 44 lists a variant generated by LLM by applying Java anonymous class transformation based on the extracted seed input program in Figure 43. As we see in Listing 62, the method to be pulled (i.e., `methodToBePulledUp()` in line 10) is surrounded by a

[72] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1533
[73] https://platform.openai.com/docs/overview
[74] https://youtrack.jetbrains.com/issue/IDEA-364110
[75] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1773
[76] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1823

```java
public class A {
    public class BaseInner {
        void innerMethodLambda(Outer outer) {
            Runnable r = () -> {
                System.out.println(outer.x);
                outer.foo();
            };
            r.run();
        }
    }
    public class Outer {
        public int x = 0;
        public void foo(){};
        public class Inner extends BaseInner {
            // Pull this method up to BaseInner
            void innerMethod() {
                innerMethodLambda(Outer.this);
            }
        }
    }
}
```

Listing 64. The generated variant by applying the Java lambda expression transformation.

```java
import com.example.one.A.Outer.Inner;
public class A {
    public class BaseInner {
        void innerMethodLambda(Outer outer) {
            Runnable r = () -> {
                System.out.println(outer.x);
                outer.foo();
            };
            r.run();
        }
        void innerMethod(Inner inner) {
                innerMethodLambda(Outer.this);//
                    syntax error
        }
    }
    public class Outer {
        public int x = 0;
        public void foo(){};
        public class Inner extends BaseInner {}
    }
}
```

Listing 65. Apply pull up method refactoring for the variant produces uncompilable program in Eclipse (2024-09) and IntelliJ IDEA (2024.2.4).

Fig. 45. One variant generated by LLM by applying the Java lambda expression transformation successfully uncover new bugs in both Eclipse (2024-09) and IntelliJ IDEA (2024.2.4).

newly created anonymous class. After performing the "Pull Up Method" refactoring in Eclipse (2024-09), it produces a refactored program that contains a syntax error. As shown in Listing 63, the pulled-up method references a method (i.e., `methodHelper()` in line 12) which is outside its visible scope. Listing 64 in Figure 45 shows a variant generated by LLM by applying the transformation of the Java lambda expression. The method to be pulled (i.e., `innerMethod()` in line 16) references a method (i.e., `innerMethodLambda(Outer outer)` in line 3) with lambda expression. Both Eclipse (2024-09) and IntelliJ IDEA (2024.2.4) cannot handle this variant correctly, thus producing a refactored program that contains a syntax error. As shown in the Listing 65 line 12, the parameter for the method `InnerMethodLambda()` remains as `Out.this` reference, which is not accessible after pulling up the method.

Instead of performing the mutation by LLM, we could also use static program analysis to operate the AST of the input program as in some metamorphic testing works [103, 105, 107]. Our collected historical bug dataset could serve as the seeds for mutation-based testing techniques. The bug triggering characteristics in Table 7 together with our five distilled findings in Section 4.4 could serve as references for the design of the mutation operators. We believe that based on our dataset and bug-triggering characteristic taxonomy, more bugs could be revealed using test generation methods.

Actionable Suggestion 2. Automated test migration for refactoring engines.

Automated test transfer or migration has been widely explored for mobile applications [5, 6, 37, 57], GUI testing [38, 71, 109], and database systems [111]. However, test migration for refactoring engines remains unexplored. Our transferability study in Section 5 has demonstrated that the input programs in the historical bug reports could be reused to uncover bugs among refactoring engines. As a preliminary study, we conducted the experiment manually. However, it would be worth exploring how to automate the migration of tests among refactoring engines because refactoring engines such as Eclipse and IntelliJ IDEA share a lot of common refactoring types, and the bug could be triggered by the same input program in different refactoring engines according to our study in Section 5. *The tests could be extracted from historical bug reports or official test suites from each refactoring engines, then a unified test suite could be constructed*

```java
public class A {
    public class BaseInner {}
    public class Outer {
        public int x = 0;
        public void foo(){};
        public class Inner extends BaseInner {
            void innerMethod() {
                System.out.println(Outer.this.x);
                Outer.this.foo();
            }
        }
    }
}
```

Listing 66. A historical bug-triggering input program.

```java
public class OuterClass {
    public class BaseTargetClass { }
    public class OriginalClass {
        public DataType memberVariable;
        public void memberMethod();
        public class NestedOriginalClass extends
              BaseTargetClass {
            void methodToBePulledUp() {
                // Method logic that accesses
                     OriginalClass's context
            }
        }
    }
}
```

Listing 67. Extracted input program template by LLM.

Fig. 46. Input program from a historical bug report (Listing 66), and its corresponding template extracted by LLM (Listing 67).

and classified by refactoring type. A framework could be developed to automate the execution of the unified test suite for different refactoring engines to uncover new bugs.

6.1.2 How our findings could benefit existing tools.

Improvement 1. Automatically extract the input program templates for ASTGen.
Finding 1 together with the analysis in Section 5.3 concludes that one of the main reasons for limiting the effectiveness and scalability of ASTGen is the limited number of predefined input program templates for just a few types of refactoring. First, ASTGen requires developers to manually create predefined templates to generate input programs. However, developers may lack the necessary knowledge or insight into the types of program structure that are more likely to expose bugs. Second, modern refactoring engines in IDEs such as IntelliJ IDEA support a wide range of refactoring types, making manual template design both labor-intensive and impractical. Third, the input programs that tend to trigger bugs can be diverse and complex for each refactoring type, which further limits the effectiveness and scalability of tools that rely on manually crafted templates. Thus, it is natural to automate the extraction of input program templates for ASTGen.

One actionable method to achieve this is to automatically extract the input program templates from the historical bug-triggering input programs in our collected dataset using LLM. As shown in Figure 46, for the input program in Listing 66 from a historical bug report[77] of Eclipse, we asked gpt-4o-mini to extract its corresponding template. During this step, we aim to get the structure template of the input program, which includes the critical program elements and structure related to the applied refactoring in the input program which are essential to trigger bugs. Listing 67 shows the extracted template, which abstracts the input program and preserves the critical structure. Based on the template, we can generate different variants with enriched context, and the refactoring applied to the input program is still applicable for the mutated variants. The extracted template could also serve as a reference for developers when designing templates, or could be dynamically combined with template-based input program generation tools, such as ASTGen, to fully automate the testing process. The template format could be adjusted through a few-shot learning. Specifically, we only need to manually extract templates for a very few number of input programs, then we could construct prompts to instruct the extraction of templates for the rest of input programs. The extracted templates could be used as input for template-based input program generation tools, such as ASTGen. The input programs could be

[77] https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1533

obtained from our collected dataset. This method could relieve developers or researchers from manual effort; meanwhile, it also extends the scalability of ASTGen, since there are various types of refactoring in our collected dataset.

Improvement 2. Design new oracles to uncover more diverse refactoring engine bugs.
Findings 3 and 11, together with the analysis in Section 5.3 indicate that existing tools still fall short to detect various bugs in the refactoring engine due to the lack of the corresponding oracles. For example, both ASTGen and SAFEREFACTOR cannot detect code comment-related bugs. SAFEREFACTOR is unable to detect comment-related bugs because it focuses solely on the functional behavior of programs during test generation using Randoop. A similar limitation also exists in ASTGen, which only designs program generators, such as field reference and class relationship generators, without incorporating any mechanism to generate or analyze code comments. As we indicated in Finding 3, existing works also do not consider the side effects that could be induced by the refactoring. For example, the breakpoints for the input program could be changed to incorrect locations after refactoring. These findings call for the design of new oracles for testing refactoring engine bugs. Our classified symptoms in Section 4.2 together with the corresponding bug reports in our dataset could serve as a basis and reference for the design of new oracles.

6.2 Implications for Developers and Researchers

Based on our study and analysis, we discuss the implications for developers and researchers of refactoring engines.

Implication for Developers. Based on Finding 1, "Extract", "Inline", and "Move" refactoring are the most error prone. Refactoring engine developers should focus on improving the reliability of these types, expanding test coverage, and considering more diverse input programs. Current testing tools only cover about one third of the refactoring types, leaving gaps that could be targeted for improvement. Finding 2 emphasizes that the most common bug symptoms are "Compile errors", "Crash", and "Behavior change". Developers should improve test oracles, particularly for detecting behavior changes, which pose significant challenges. Furthermore, accounting for side effects such as warnings and refactoring availability (Finding 3) should also be integrated into error-checking processes to ensure consistency. Finding 4 highlights the prominence of incorrect transformations as a root cause of bugs. Developers should work towards diverse input programs and leverage data-driven automation to minimize these issues. Furthermore, enhancing flow analysis and type resolution mechanisms (Finding 5) and automating the inference of preconditions (Finding 6) will help mitigate errors. Given that refactoring involving modern Java language features triggers a significant number of bugs (Finding 8a), refactoring engines should improve their handling of language-specific features like lambda expression, generic, and Enum (Finding 8b). These features present complexities in type inference and flow analysis, which should be better addressed in future engine iterations. Finding 9 shows that complex class structures, such as inner and anonymous classes, are prone to triggering bugs. Similarly, annotations (Finding 10) add complexity because of their semantic meaning and wide-ranging effects. Refactoring engines should be enhanced to handle these constructs more effectively, particularly by improving transformations that respect class and annotation intricacies. Finding 11 emphasizes the challenges posed by comments interleaved with code, where dependencies between code and comments can lead to semantic gaps. Developers should explore new methods to make refactoring tools more comment-aware, ensuring that the intended meaning of both code and comments is preserved during refactoring.

Implication for Researchers. Based on Finding 1, future research should explore improved testing techniques that focus on the most error-prone refactoring. These techniques should go beyond existing tools to cover a wider range of refactoring types, leveraging the findings as a basis for designing more effective tests. Finding 2 points out the challenges in detecting behavior change bugs. Researchers should focus on how to identify behavior changes effectively and efficiently. In addition, incorporating side effects, such as refactoring availability or warning messages, into bug

detection strategies could provide a broader perspective on refactoring correctness. As mentioned in Findings 4 and 5, automated, data-driven approaches are a promising direction to improve the testing of the refactoring engine. Researchers could investigate large language models (LLMs) to generate diverse input programs. Finding 7 reveals that most of the refactoring engine bugs could be triggered by the default configuration, thus relieving researchers from testing configuration-related refactoring bugs. Findings 8a and 8b highlight that language features, particularly lambda expressions and generics, significantly contribute to refactoring bugs. Researchers should incorporate new language features into their testing techniques to ensure a better coverage of modern programming language features. Findings 9 and 10 suggest that complex class relationships and annotations are common triggers for refactoring bugs. Future research could explore how refactoring engines can be improved to handle them better. Based on Finding 11, there is a gap in research on comment-aware refactoring. Researchers should investigate new methodologies for creating tools that can bridge the semantic gap between code and comments, enabling more robust refactoring that considers code-comment dependencies. Based on Finding 12 and the results in RQ5, more effective testing tools could be designed. For example, by analyzing the relationship between the characteristics of the input program and the types of refactoring, researchers can investigate which characteristics of the input program are more likely to result in certain refactoring failures. Thus, they could design tools to generate or prioritize the input program that has certain characteristics for some refactoring to improve the effectiveness and efficiency of the tests.

7 Threats to Validity

We identify the following threats to the validity of this paper:

Internal. The internal threat to validity mainly lies in our manual classification and labeling of refactoring engine bugs, which may have subjective bias or errors. To reduce this threat, we refer to previous studies [59, 70, 102], and then adopt an open-coding scheme to derive the taxonomies to fit refactoring engine bugs. During the labeling process, the first two authors independently labeled the bugs and any disagreement was discussed at a meeting until a consensus was reached.

External. The external threat to validity lies mainly in the dataset used in our study. To reduce this threat, we systematically collected refactoring engine bugs as presented in Section 3. To guarantee the generalizability of our study, we used three popular refactoring engines as subjects and studied 518 bugs to balance the effort of manual analysis and the study scale. The keywords used for searching bug reports could affect our collected dataset. However, our keywords (i.e., “refactoring” and “refactor”) were able to retrieve a wide range of refactoring engine bug reports covering various refactoring types, as shown in Table 4. Our analysis focuses on Java programs and Java refactoring engines. We chose Java because it is one of the most widely used programming languages [52]. We chose Java-based refactoring engines because they support the most diverse refactoring types and are widely used. For example, Eclipse CDT provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform; however, at the time of our study, its latest version 2024-12 (4.34.0) only provides support for six types of refactoring (i.e., Rename, Extract Local Variable, Extract Constant, Extract Function, Toggle Function, and Hide Method). The same version of Java-based refactoring engines (i.e., JDT) provides 24 types of refactoring, which could give us a broader scope for our study. Since the underlining implementation of the refactoring engine for different programming languages shares a similar workflow (i.e., all engines need to check preconditions and rewrite AST to make code transformations) as we illustrated in Figure 1, the bug symptoms (RQ2) and root causes (RQ3) are generally applicable as in our study. Regarding the characteristics of the bug-triggering input program (RQ4), we categorized the Java language-specific features in Table 7 in Section 4.4 as row T1 — “Language Features”, and the other categories (i.e., T2 – T8 in Table 7) can be generalized to

other programming languages. Since there is a limited support of refactoring types for other program languages (e.g., six versus 24 for C/C++ and Java), for RQ1 (error-prone refactoring type), the error-prone refactoring type distribution might be different, we leave RQ1 (error-prone refactoring type) with respect to refactoring engines supporting program languages beyond Java as future work.

8 Related Work

Detecting Refactoring Engine Bugs. There are several techniques to test refactoring engine bugs in the literature [10, 21, 46, 64–68, 81]. For example, Daniel et al. [10] developed ASTGen, a widely used Java test program generation tool, to test refactoring engines. Soares et al. [65] proposed SAFEREFACTOR, which is based on random tests to ensure that the refactoring is behavior-preserving. Soares et al. [46, 64, 67, 68] proposed testing refactoring engines based on SAFEREFACTOR. Meanwhile, Gligoric et al. [21] tested Eclipse refactoring engines for Java (JDT) and C (CDT) in an end-to-end approach in real software projects. Given a set of projects, they apply refactoring in many places and collect failures where the refactoring engine throws an exception or produces refactored code that does not compile. Eric et al. [34] analyzed the existing period, fixing ratio, and duplication percentage of refactoring bugs of Java Development Tools (JDT). To validate the effectiveness of the refactoring detection tools (i.e., RefactoringMiner [76] and RefDiff [60]), Leandro et al. [35] first apply refactoring using a popular IDE (i.e., Eclipse) and then run these two refactoring detection tools to check whether they can detect the transformations performed by the IDE. Pablo et al. [74] studied the preservation of the state of the program in live environments. Anquetil et al. [2] analyzed atomic transformations that could be reused among different refactoring. Balša [112] proposed a new architecture of the refactoring engine. Unlike existing research in refactoring bug detection, our work focuses on studying the characteristics of refactoring engine bugs to facilitate testing and debugging. Current approaches may be ineffective as (1) they randomly generate test programs or apply refactoring without any guidance, and (2) lack a general and in-depth understanding of bugs in the refactoring engines. Therefore, as the first empirical study on refactoring engine bugs, we believe that our work helps design effective testing and debugging methods based on our findings.

Empirical Studies on Bugs. Researchers have conducted various empirical studies to understand the characteristics of bugs, including compilers [59, 70, 100, 110], Android [7, 39, 99], static analyzers [104, 106, 108], etc. For example, Sun et al. [70] conducted an empirical study to analyze the duration, priority, fixes, test cases, and locations of GCC and LLVM bugs. Zhong et al. [110] study historical compiler bug reports and enrich compiler testing with real programs from bug reports. Xiong et al. [99] study functional bugs in Android applications. Zhang et al. [106] study annotation-induced faults for static analyzers like PMD. Unlike previous studies, we target refactoring engine bugs. To the best of our knowledge, we are the first to characterize refactoring engine bugs focusing on various perspectives (symptoms, root causes, triggering conditions). Our study helps to gain a better understanding of the bugs in the refactoring engine, which can potentially benefit developers of refactoring engines and researchers in the detection of refactoring engine bugs. Based on our findings and dataset, our transferability study has revealed more than 100 new bugs in the latest version of the refactoring tools.

9 Conclusion

As integral components of modern Integrated Development Environments (IDEs), refactoring engines can alleviate the burden of development and optimization by assisting developers in restructuring their code. However, it is inevitable for these engines to have bugs, as in traditional software systems. Their bugs could be propagated to the system refactored by them and produce unexpected, even dangerous behaviors during real-world usage. Therefore, it is necessary to

guarantee the quality of refactoring engines. In this paper, we conducted the first comprehensive study to understand refactoring engine bugs to promote the design of effective bug detection and debugging techniques. Specifically, we manually studied 518 bugs from three popular refactoring engines, identified several root causes, bug symptoms, and input program characteristics, and obtained 12 major findings. Based on these findings, we provide a set of guidelines for refactoring engine bug detection and debugging. Our transferability study reveals 134 new and unique bugs in the latest version of the refactoring tools. Among the 22 bugs that we submitted, 11 bugs are confirmed by their developers, and seven of them have already been fixed.

Data Availability. The data is available at [79].

Acknowledgments

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants RGPIN-2024-04301.

References

[1] Eman Abdullah AlOmar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2022. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering* 18, 1 (2022), 105–135.

[2] Nicolas Anquetil, M Campero, Stéphane Ducasse, J.-P Sandoval, and Pablo Tesone. 2022. Transformation-based Refactorings: a First Analysis. In *IWST 22 - International Workshop of Smalltalk Technologies*. Publisher, Novisad, Serbia, –. https://inria.hal.science/hal-03752247

[3] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. 2018. PreInfer: Automatic inference of preconditions via symbolic analysis. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, IEEE, New York, NY, USA, 678–689.

[4] Paul Becker, Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Boston, MA.

[5] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 54–65. https://doi.org/10.1109/ASE.2019.00016

[6] Benyamin Beyzaei, Saghar Talebipour, Ghazal Rafiei, Nenad Medvidovic, and Sam Malek. 2024. Automated Test Transfer Across Android Apps Using Large Language Models. *arXiv preprint arXiv:2411.17933* 1, 1 (2024), xx–yy.

[7] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, IEEE, Piscataway, NJ, USA, 133–143.

[8] Xiaye Chi, Hui Liu, Guangjie Li, Weixiao Wang, Yunni Xia, Yanjie Jiang, Yuxia Zhang, and Weixing Ji. 2023. An Automated Approach to Extracting Local Variables. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 313–325.

[9] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, Springer, Berlin, Heidelberg, 128–148.

[10] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, Dubrovnik, Croatia, 185–194.

[11] Xiaoting Du, Zheng Zheng, Lei Ma, and Jianjun Zhao. 2021. An empirical study on common bugs in deep learning compilers. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, IEEE, Piscataway, NJ, USA, 184–195.

[12] Bart Du Bois, Serge Demeyer, and Jan Verelst. 2004. Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*. IEEE, IEEE, Piscataway, NJ, USA, 144–151.

[13] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*. Publisher, Address, 779–790.

[14] Aryaz Eghbali and Michael Pradel. 2020. No strings attached: An empirical study of string-related software bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Publisher, Address, 956–967.

[15] Eclipse Foundation. 2024. *Eclipse*. Organization. http://www.eclipse.org/

[16] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Address.

[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Publisher, Address, 416–419.

[18] Garcia. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. Publisher, Address, 385–396.

[19] Kely M Garcia. 2007. *Testing the Refactoring Engine of the NetBeans IDE*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[20] GitHub. 2022. *GitHub APIs*. GitHub. https://docs.github.com/en/rest?apiVersion=2022-11-28
[21] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. 2013. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*. Springer, Publisher, Address, 629–653.
[22] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Publisher, Address, 1303–1313.
[23] Shuo Hong, Hailong Sun, Xiang Gao, and Shin Hwei Tan. 2024. Investigating and Detecting Silent Bugs in PyTorch Programs. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Publisher, Address, 272–283.
[24] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. Publisher, Address, 510–520.
[25] Dmitry Jemerov. 2008. Implementing refactorings in IntelliJ IDEA. In *Proceedings of the 2nd Workshop on Refactoring Tools*. Publisher, Address, 1–2.
[26] Jetbrains. 2023. *Java IDE Usage Stats*. Jetbrains. https://www.jetbrains.com/lp/devecosystem-2023/java/#java_ide
[27] JetBrains. 2024. *IntelliJ IDEA*. Organization. http://www.jetbrains.com/idea/
[28] JetBrains. 2024. *IntelliJ IDEA Community Edition*. JetBrains. https://github.com/JetBrains/intellij-community
[29] JetBrains. 2024. *IntelliJ IDEA Issue Tracker*. JetBrains. https://youtrack.jetbrains.com/issues/IDEA
[30] Bishoksan Kafle, Graeme Gange, Peter J Stuckey, Peter Schachte, and Harald Søndergaard. 2021. Transformation-enabled precondition inference. *Theory and Practice of Logic Programming* 21, 6 (2021), 700–716.
[31] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23, 2009 (2009), 2009.
[32] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.
[33] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2018. An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information and Software Technology* 93 (2018), 186–199.
[34] Eric Lacker, Jongwook Kim, Akash Kumar, Lipika Chandrashekar, Srilaxmi Paramaiahgari, and Jasmine Howard. 2021. Statistical analysis of refactoring bug reports in eclipse bugzilla. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, Publisher, Address, 9–13.
[35] Osmar Leandro, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, and Alessandro Garcia. 2022. A Technique to Test Refactoring Detection Tools. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. Publisher, Address, 188–197.
[36] Tao Li and Yang Zhang. 2024. Multilingual code refactoring detection based on deep learning. *Expert Systems with Applications* X, Y (2024), 125164.
[37] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Publisher, Address, 42–53. https://doi.org/10.1109/ASE.2019.00015
[38] Jun-Wei Lin and Sam Malek. 2022. GUI Test Transfer from Web to Android. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Publisher, Address, 1–11. https://doi.org/10.1109/ICST53961.2022.00011
[39] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. 2017. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, Publisher, Address, 2–13.
[40] Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Publisher, Address, 908–920.
[41] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.
[42] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. Publisher, Address, 385–396.
[43] Mary L McHugh. 2013. The chi-square test of independence. *Biochemia medica* 23, 2 (2013), 143–149.
[44] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.
[45] Microsoft. 2021. *Java Language Support for Visual Studio Code*. Organization. https://github.com/redhat-developer/vscode-java
[46] Melina Mongiovi. 2016. Scaling testing of refactoring engines. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. Publisher, Address, 15–17.
[47] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2017. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering* 44, 5 (2017), 429–452.
[48] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering* 44, 5 (2018), 429–452. https://doi.org/10.1109/TSE.2017.2693982
[49] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*. Springer, Publisher, Address, 552–576.
[50] Oracle. 2024. *Record Classes*. JetBrains. https://docs.oracle.com/en/java/javase/22/language/records.html

[51] Oracle. 2024. *Record Patterns*. Organization. https://docs.oracle.com/en/java/javase/22/language/record-patterns.html

[52] Aleksandra Orlowska, Christos Chrysoulas, Zakwan Jaroucheh, and Xiaodong Liu. 2021. Programming Languages: A Usage-based Statistical Analysis and Visualization. In *Proceedings of the 4th International Conference on Information Science and Systems* (Edinburgh, United Kingdom) *(ICISS '21)*. Association for Computing Machinery, New York, NY, USA, 143–148. https://doi.org/10.1145/3459955.3460614

[53] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[54] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[55] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Publisher, Address, 125–136.

[56] Michael Petito. 2007. Eclipse refactoring. *http://people. clarkson. edu/˜ dhou/courses/EE564-s07/Eclipse-Refactoring. pdf* 5 (2007), 2010.

[57] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 284–295. https://doi.org/10.1145/3293882.3330575

[58] Guogen Shan and Shawn Gerstenberger. 2017. Fisher's exact approach for post hoc analysis of a chi-squared test. *PloS one* 12, 12 (2017), e0188709.

[59] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. Publisher, Address, 968–980.

[60] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.

[61] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. Publisher, Address, xx–yy.

[62] Indy PSC Silva, Everton LG Alves, and Wilkerson L Andrade. 2017. Analyzing automatic test generation tools for refactoring validation. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. IEEE, Publisher, Address, 38–44.

[63] Indy P.S.C. Silva, Everton L.G. Alves, and Wilkerson L. Andrade. 2017. Analyzing Automatic Test Generation Tools for Refactoring Validation. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. Publisher, Address, 38–44. https://doi.org/10.1109/AST.2017.9

[64] Gustavo Soares. 2010. Making program refactoring safer. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. Publisher, Address, 521–522.

[65] Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Márcio Cornélio. 2009. Saferefactor-tool for checking refactoring safety. *Tools Session at SBES* N/A, N/A (2009), 49–54.

[66] Gustavo Soares, Rohit Gheyi, Tiago Massoni, Márcio Cornélio, and Diego Cavalcanti. 2009. Generating unit tests for checking refactoring safety. In *Brazilian Symposium on Programming Languages*, Vol. 1175. Publisher, Address, 159–172.

[67] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. 2011. Identifying overly strong conditions in refactoring implementations. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Publisher, Address, 173–182.

[68] Gustavo Soares Soares. 2012. Automated behavioral testing of refactoring engines. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. Publisher, Address, 49–52.

[69] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. 2008. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. Publisher, Address, 653–662.

[70] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. Publisher, Address, 294–305.

[71] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI Test Migration Across Mobile Platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Publisher, Address, 756–767. https://doi.org/10.1109/ASE51524.2021.9678643

[72] Honghao Tan, Haibo Wang, Diany Pressato, Yisen Xu, and Shin Hwei Tan. 2025. Automated Harmfulness Testing for Code Large Language Models. *arXiv preprint arXiv:2503.16740* X, Y (2025), xx–yy.

[73] NetBeans Team. 2024. *NetBeans*. Organization. http://netbeans.org/

[74] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. 2020. Preserving instance state during refactorings in live environments. *Future Generation Computer Systems* 110 (2020), 1–17. https://doi.org/10.1016/j.future.2020.04.010

[75] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[76] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* XX, YY (2020), xx–yy.

[77] Michael Wahler, Uwe Drofenik, and Will Snipes. 2016. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Publisher, Address, 493–501.

[78] Haibo Wang. 2024. *Found bugs.* Concordia University. https://docs.google.com/spreadsheets/d/16RNtejiYSOeiLpxzk-9QJUKcdUEs199_zwkI8VMkS8Q/edit?usp=sharing

[79] Haibo Wang. 2024. *Our repository.* Organization. https://github.com/HaiboWang1992/RETester

[80] Haibo Wang, Zezhong Xing, Chengnian Sun, Zheng Wang, and Shin Hwei Tan. 2025. Towards Diverse Program Transformations for Program Simplification. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE015 (June 2025), 23 pages. https://doi.org/10.1145/3715730

[81] Haibo Wang, Zhuolin Xu, and Shin Hwei Tan. 2025. Testing Refactoring Engine via Historical Bug Report driven LLM. *arXiv preprint arXiv:2501.09879* 1, 1 (2025), xx–yy.

[82] Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility Issues in Deep Learning Systems: Problems and Opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Publisher, Address, 476–488.

[83] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., Address, 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[84] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, Publisher, Address, 53–64.

[85] Hsu Myat Win, Haibo Wang, and Shin Hwei Tan. 2023. Towards Automated Detection of Unethical Behavior in Open-Source Software Projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Publisher, Address, 644–656.

[86] Hsu Myat Win, Haibo Wang, and Shin Hwei Tan. 2023. Towards Automated Detection of Unethical Behavior in Open-Source Software Projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 644–656. https://doi.org/10.1145/3611643.3616314

[87] Real world developers. 2022. *Improve the Safety of "Extract Local Variable" Refactorings by Identifying the Side Effect of Selected Expression.* Organization. https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/348

[88] Real world developers. 2024. *AI Assistant.* Organization. https://www.jetbrains.com/help/idea/use-prompts-to-explain-and-refactor-your-code.html#ai-suggest-refactoring

[89] Real world developers. 2024. *Bug 544992 - [10][refactoring] convert local variable to field not working for 'var'.* Organization. https://bugs.eclipse.org/bugs/show_bug.cgi?id=544992

[90] Real world developers. 2024. *CCE with Extract Class refactoring on a class with comment.* Organization. https://bugs.eclipse.org/bugs/show_bug.cgi?id=408124

[91] Real world developers. 2024. *Comment deleted on 'Move type to new file' refactoring.* Organization. https://bugs.eclipse.org/bugs/show_bug.cgi?id=305103

[92] Real world developers. 2024. *"Encapsulate fields" refactoring removes my comments.* Organization. https://youtrack.jetbrains.com/issue/IDEA-124747/Encapsulate-fields-refactoring-removes-my-comments

[93] Real world developers. 2024. *Extract Method refactoring produces refactored program contains syntax error.* Organization. https://youtrack.jetbrains.com/issue/IDEA-354122/Extract-Method-refactoring-produces-refactored-program-contains-syntax-error

[94] Real world developers. 2024. *Extract Parameter Object creates wrong Javadoc.* Organization. https://youtrack.jetbrains.com/issue/IDEA-108187/Extract-Parameter-Object-creates-wrong-Javadoc

[95] Real world developers. 2024. *Improve the Safety of "Extract Local Variable" refactorings by identifying statements that may change the value of the extracted expressions.* JetBrains. https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/432

[96] Real world developers. 2024. *Refactor > Extract Local Variable and Extract Constant not working on Text Block.* Organization. https://bugs.eclipse.org/bugs/show_bug.cgi?id=551002

[97] Real world developers. 2024. *RefactoringMiner.* Organization. https://github.com/tsantalis/RefactoringMiner

[98] Guanping Xiao, Zheng Zheng, Bo Jiang, and Yulei Sui. 2020. An Empirical Study of Regression Bug Chains in Linux. *IEEE Transactions on Reliability* 69, 2 (2020), 558–570. https://doi.org/10.1109/TR.2019.2902171

[99] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Publisher, Address, 1319–1331.

[100] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. Publisher, Address, 3655–3672.

[101] Zhuolin Xu, Qiushi Li, and Shin Hwei Tant. 2025. Understanding and Enhancing Attribute Prioritization in Fixing Web UI Tests with LLMs. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Publisher, Address, 326–337. https://doi.org/10.1109/ICST62969.2025.10989008

[102] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. Publisher, Address, 283–294.

[103] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software*

Engineering (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 237–249. https://doi.org/10.1145/3611643.3616272

[104] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Publisher, Address, 237–249.

[105] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proc. ACM Softw. Eng.* 1, FSE, Article 33 (July 2024), 23 pages. https://doi.org/10.1145/3643759

[106] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 722–744.

[107] Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1772–1784. https://doi.org/10.1145/3650212.3680398

[108] Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1772–1784. https://doi.org/10.1145/3650212.3680398

[109] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3368089.3409708

[110] Hao Zhong. 2022. Enriching compiler testing with real program from bug report. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Publisher, Address, 1–12.

[111] Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proc. ACM Manag. Data* 2, 6, Article 253 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698829

[112] Balša Šarenac, Nicolas Anquetil, Stéphane Ducasse, and Pablo Tesone. 2024. A new architecture reconciling refactorings and transformations. *Journal of Computer Languages* 80 (2024), 101273. https://doi.org/10.1016/j.cola.2024.101273