

Testing Refactoring Engine via Historical Bug Report driven LLM

Haibo Wang, Zhuolin Xu, Shin Hwei Tan*

Department of Computer Science & Software Engineering, Concordia University, Montreal, Canada
haibo.wang@mail.concordia.ca, zhuolin.xu@mail.concordia.ca, shinhwei.tan@concordia.ca

Abstract—Refactoring is the process of restructuring existing code without changing its external behavior while improving its internal structure. 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. Similar to traditional software systems such as compilers, refactoring engines may also contain bugs that can lead to unexpected behaviors. In this paper, we propose a novel approach called RETESTER, a LLM-based framework for automated refactoring engine testing. Specifically, by using input program structure templates extracted from historical bug reports and input program characteristics that are error-prone, we design chain-of-thought (CoT) prompts to perform refactoring-preserving transformations. The generated variants are then tested on the latest version of refactoring engines using differential testing. We evaluate RETESTER on two most popular modern refactoring engines (i.e., ECLIPSE, and INTELLIJ IDEA). It successfully revealed 18 new bugs in the latest version of those refactoring engines. By the time we submit our paper, seven of them were confirmed by their developers, and three were fixed.

Index Terms—Test generation, Refactoring, Refactoring engine, Bug detection

I. 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, yet improves its internal structure [1]. Refactoring has been well-studied as a way to improve software quality [2], [3] as well as an effective way to facilitate software maintenance and evolution [4], [5]. During software development, developers might perform refactoring manually, which is error-prone and time-consuming, or with the help of tools that automate or semi-automate the activities related to the refactoring process [6], [7]. Refactoring automation tools like the built-in refactoring engines in IDEs (e.g., ECLIPSE [8], and INTELLIJ IDEA [9]) have been widely used to facilitate software maintenance.

Despite the active development of refactoring engines, they can be buggy. These bugs can silently change the program behaviors, produce uncompileable programs, or induce inconsistencies in the modified code [10]. To ensure the reliability of refactoring engines, several techniques have been proposed to identify Refactoring Engine Bugs (REBs) via input programs generation, including (1) template-based input program generation technique [11] that relies on manually crafted imperative generators, and (2) SAFEREFACITOR-based tools [12], [13], [14], [15], [16], [17], [18] that relies that random test gen-

eration. However, there are several limitations in these tools that hinder their effectiveness in identifying REBs. Firstly, template-based input program generation techniques like AST-GEN [11] usually require developers to manually write a set of predefined templates for generating input programs. However, developers may not have sufficient knowledge nor insights about the types of program structures that are more likely to trigger bugs. Moreover, modern refactoring engines in IDE like INTELLIJ IDEA support a variety of refactoring types which makes manually designing templates labor-intensive and impractical. Secondly, SAFEREFACITOR-based tools [12], [13], [14], [15], [16], [17] rely on automatic test generation tools to identify refactoring bugs without considering the bug-triggering ability of the input programs (e.g., input programs contain lambda expression or anonymous class are more error-prone during refactoring), thus leading to low effectiveness.

In recent years, large language model (LLM) trained on programming languages as well as natural languages has shown a great potential to support various software engineering tasks like program repair, code summarization, and code review automation. Prior studies [19], [20], [21], [22] have explored the use of LLM for refactoring activities. However, the feasibility of using LLM to improve the robustness and reliability of the refactoring engines has been under-explored. Given the importance of identifying REBs and the limitations of existing approaches, in this paper, we propose a testing approach for refactoring engines called RETESTER that leverages historical bug reports together with the bug-triggering input program characteristics, which can be generalizable to support diverse refactoring types. In particular, RETESTER leverages historical bug reports, to extract refactoring information. Then, RETESTER utilizes LLM combined with prompts engineering to perform Refactoring-preserving Transformations (RPTs). Specifically, by using the input program characteristics that are more error-prone as mutation rules and input program structure templates, we design chain-of-thought (CoT) prompts to perform refactoring-preserving transformations. The generated variants are then tested on the latest version of refactoring engines by applying the same refactoring as in the seed input program. Finally, we use differential testing to find any inconsistencies, and manually inspect each inconsistency before submitting issues to the refactoring engine developers.

Our proposed workflow can benefit refactoring engine testing from two perspectives: First, we leverage the power of LLM to generate diverse bug-triggering input programs based

on historical bug reports, which can serve as the first step towards LLM-based testing for refactoring engines. Second, by extracting diverse historical bug-triggering input program structures, template-based techniques (e.g., ASTGen [11] for refactoring engine testing, JAttack [23] and LeJit [24] for compiler testing) can take these error-prone templates as references for designing their templates, thus improving their effectiveness. In summary, this paper makes the following contributions:

- We mined and propose a new dataset that contains human-labeled and high-quality historical bug reports, together with 167 compilable Java programs extracted from these bug reports. Programs in our dataset can be used as the seed inputs for future research on testing software systems that takes Java programs as input (e.g., refactoring engines and compilers).
- We propose a novel LLM-based refactoring engine bug detection approach that (1) automatically mines historical bug reports and extracts high-quality bug-triggering input programs via LLM, (2) performs refactoring-preserving mutations of input programs to obtain similar bug-triggering programs, and (3) leverages prompt template that extracts refactoring information (e.g., refactoring types, and program locations to perform refactoring) from historical bug reports. To the best of our knowledge, we conduct the first systematic study that applies LLM for testing refactoring engine bugs. We open-source our data to facilitate future research in refactoring engine testing [25].
- We conduct experiments on two popular refactoring engines (i.e., ECLIPSE, and INTELLIJ IDEA). As a result, we have found 18 new bugs in the latest version. By the time we submit our paper, seven bugs have been confirmed by their developers, three have been fixed.

II. BACKGROUND

A. Refactoring Engine

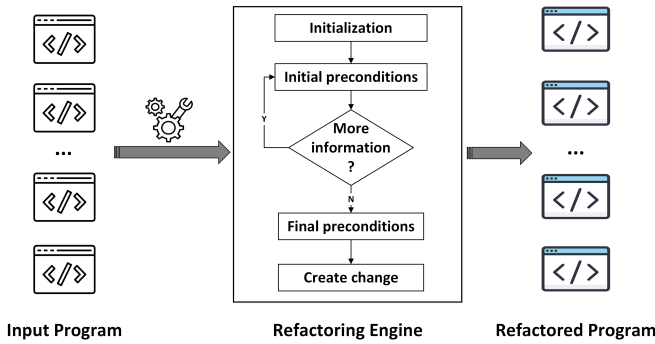


Fig. 1: The general workflow of a refactoring engine.

Refactoring engines, critical components of Integrated Development Environments (IDEs) like ECLIPSE, and INTELLIJ IDEA, could automate code restructuring and optimization. Despite varied details and integrations, their workflows share

core similarities. The general workflow, depicted in Figure 1, starts with an input program and configuration settings for a specific refactoring and results in the refactored program. The process includes: (1) Initialization, where the engine prepares for refactoring by setting up context and identifying target elements; (2) Initial Conditions Checking, which verifies prerequisites and potential conflicts to ensure refactoring can proceed—if not, errors or warnings are issued; (3) Additional Information Gathering, where the engine collects further inputs or configuration details necessary for the refactoring; (4) Final Conditions Checking, which analyzes the proposed changes, checking for conflicts and verifying if the refactoring preserves behavior and maintains quality—if issues are found, warnings or errors are generated; and finally, (5) Create Change, where the actual code modifications are computed and implemented, including updating declarations, method invocations, associated documentation, and etc. Each stage is designed to ensure that refactoring is executed correctly, maintaining behavior preservation and code syntax.

B. Refactoring Engine Testing

Several techniques have been proposed to identify refactoring engine bugs [11], [12], [13], [14], [18]. Specifically, Daniel et al. [11] proposed ASTGen, a template-based Java input program generation tool for refactoring engines testing. Soares et al. [12], [13], [14] proposed a series of works based on SAFEREFACATOR, which relies on random tests generation technique, Randoop, to test the behavior-preserving of refactoring. Gligoric et al. [18] tested Eclipse Java (JDT) and C (CDT) refactoring engines in an end-to-end approach on real software projects. Given a set of projects, they randomly apply refactoring in some program elements and collect failures using differential testing. Eric et al. [26] analyzed the existing period, fixing ratio, and duplication percentage of bugs in ECLIPSE JAVA DEVELOPMENT TOOLS (JDT). Existing works share the following limitations: (1) they are unaware of the bug-triggering ability of the input programs, thus producing or testing on input programs that are less error-prone. (2) manually designing templates for diverse refactoring types and scenarios is time-consuming and labor-intensive, which hinders the scalability. Different from previous works, our work generate refactoring-preserving variant input programs by incorporating the error-prone input program characteristics together with the historical bug-triggering code information. By incorporating the error-prone input program characteristics with the historical bug-triggering input program templates through the refactoring-preserving transformation (Definition 1), new bugs are revealed in the newest version of refactoring engines. Besides, RETESTER could tolerant diverse refactoring types without the need of manual designing templates, making it scalable, flexible, and generalizable. To our knowledge, this is the first work demonstrating that LLM can easily perform history-driven testing to the challenging domain of refactoring engine.

C. Motivation Example

[Bug][Pull Up Refactoring] Pull up method refactoring for method in the inner class fails #1533

Closed RETester66 opened this issue on Jul 22 · 0 comments · Fixed by #1590



Fig. 2: Screenshot of an ECLIPSE historical bug report [27].

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

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
            }
        }
    }
}

public class A {
    public class BaseTargetClass {}
    public class OriginalClass {
        public int data = 20;
        public void memberMethod() {}
        public class NestedOriginalClass extends
            BaseTargetClass {
            void setup() {
                new BaseTargetClass() {
                    void methodToBePulledUp() {
                        System.out.println("Anonymous
                        Class Method: " + data);
                    }
                }
            }
        }
    }
}
```

Fig. 3: A bug-triggering input program extracted from historical bug reports (top), extracted template by LLM (middle), and one of its refactoring-preserving variants after applying the Java anonymous class transformation (bottom).

Figure 2 shows the screenshot of one refactoring engine historical bug report from ECLIPSE [27]. As stated in the bug report, refactoring engine of ECLIPSE fails to resolve the method in a inner class referring to an outer class field when performing the pull up method refactoring, thus producing an uncompileable refactored program. Figure 3 lists the Java

input program (top) extracted from the above historical bug report. Specifically, when applying pull up method refactoring for the method “innerMethod()”, the refactoring could be successfully performed without any warning message or exception, however, ECLIPSE would produce an refactored program contains syntax error, thus making the original program or project uncompileable. ECLIPSE developers have fixed this issue by adding a more reliable dependency analysis to resolve the method in inner class when performing pull up method refactoring [28].

To challenge refactoring engines under more diverse input programs, our tool generates input programs by incorporating the error-prone input program characteristics and historical bug-triggering input programs from bug reports, which successfully reveals 18 new bugs in the latest version of refactoring engines. Specifically, we first construct a dataset containing historical refactoring engine bug reports by mining from refactoring engine bug-tracking systems using keywords (i.e., “refactoring” and “refactor”). To remove the irrelevant bug reports and keep the bug reports that fulfill our criteria (e.g., the bug report should contain input program and reproduce steps), a systematic manual classification and labeling process is performed. Then, to obtain the bug reports which contain compileable input program, we leverage the LLM with few-shot-learning to extract input programs from bug reports and compile them under JVM. The reasons for filtering the compileable input programs are two folds: first, ECLIPSE refactoring can only be performed on the input programs contain no syntax errors, however, some input programs in the historical bug reports could be incomplete or not syntax error free. Second, those input programs are served as the seeds for the following up mutations, uncompileable seeds could result in variants contains syntax errors, which could result in invalid input programs for ECLIPSE. After getting the historical bug reports containing compileable input programs as the seeds. We further ask the LLM to extract refactoring information, like refactoring type, input program, and symptom, from those seed bug reports using few-shot-learning by feeding reports’ content. Those information will be used in the following up step. Meanwhile, we also ask the LLM to extract the input program structure template, which is the template to represent the structure of the input program. The purposes to extract input program template structure are following: (1) input program structure template keeps the historical bug-triggering input program’s structure, meanwhile, it is abstracted by removing the detailed code logic related with current refactoring, thus making a larger search space while mutating the program. For example, the extracted template for input program on the top is listed in the middle of Figure 3. Compared with the input program, the code logic inside the method “innerMethod()” is removed, so, when mutating the input program, the code mutation will not be limited on existing code logic. (2) Our extracted templates could benefit template-based testing techniques like ASTGEN since it could serve as the basis or reference. After the refactoring information extraction, we apply the refactoring-preserving transformation (Definition 1)

by incorporating the bug-triggering input program characteristics (Table IV) together with the templates extracted, thus generating variants that both contain more diverse context and are error-prone. For instance, according to existing study [10], input programs with Java anonymous class are more likely to trigger refactoring engine bugs. By constructing chain-of-thought prompts, we instruct LLM to generate refactoring-preserving variants by incorporating the anonymous class with the extracted template. The program in the bottom of Figure 3 shows one of the refactoring-preserving variants generated by our tool that successfully reveal one new bug in the last version (2024-09) of ECLIPSE based on the seed program in the top. For this variant, ECLIPSE fails to resolve the method in an anonymous class when performing the pull up refactoring, producing an refactored program contains syntax error. We have submitted this bug to the ECLIPSE developers [29].

III. METHODOLOGY

A. Overview

Figure 4 provides a overview of RETESTER. Initially, we construct our dataset by gathering historical bug reports from the issue tracker systems of ECLIPSE and INTELLIJ IDEA, specifically targeting refactoring engine issues. To ensure relevance, we manually label and classify each report, discarding those unrelated to bugs, such as feature requests. We then employ a Large Language Model (LLM) with few-shot-learning to extract and compile input programs from these reports using a JVM compiler. Only reports containing compilable input programs are retained. Next, the LLM extracts crucial refactoring information from these reports, such as the type of refactoring, procedural details, and input program structure templates. Based on this information, we create mutation prompts reflecting the extracted refactoring details together with the error-prone input program characteristics, as detailed in Table IV. These prompts are fed into the LLM to generate diverse mutation variants that preserve the applicability of the original refactoring on the seed input program. Each variant undergoes the same refactoring process in the refactoring engine being tested, and we employ differential testing to determine whether these variants expose any bugs.

B. Dataset Construction

1) *Mining historical issues:* This step aims to collect a broad range of representative refactoring engine bugs. In this study, we target the two most popular refactoring engines as subjects, including (1) JAVA DEVELOPMENT TOOLS (JDT) from ECLIPSE, and (2) the Java refactoring component of INTELLIJ IDEA. There are diverse categories of bug reports with different purposes, such as feature requests and questions. Hence, we need to identify the bug reports related with refactoring engine bugs only. Specifically, following existing studies [30], [31], we collect bug reports whose title or discussions contain at least one refactoring bug-relevant keyword (i.e., “refactoring” and “refactor”) before the time of our study (July 2024). We focus on bugs that are fixed and not duplicated. Specifically, we consider a bug is fixed if its

TABLE I: The prompt template used to extract input program from historical bug reports.

You are a software testing expert. I will give you some historical refactoring engines bug reports in the following conversations, you need to extract the input programs together with their corresponding class names from the bug report. The extracted information should be in JSON format, you should only return me the extracted input program in the json format, not any natural language. I will give you examples following: **{Example}**

“Resolution” field is set to “FIXED” and the “Status” field is set to “RESOLVED”, “VERIFIED” or “CLOSED” for the ECLIPSE bug in Bugzilla. After Apr 2022, ECLIPSE started to migrate their issue trackers to GitHub. To get a complete list of bug reports, we also crawled issues from their GitHub repositories using the GitHub APIs [32]. For each repository, we search for the fixed issues with the same keywords. For INTELLIJ IDEA, we collect fixed bugs from its issue tracker using the same keywords. For the issues marked as duplicates by the refactoring engine developers, we will not include them.

2) *Classification and Labeling Process:* As it is too time-consuming to manually analyze all bugs, we only kept the top 1000 crawled results sorted by fix time for each of our studied refactoring engine. To remove irrelevant issues with refactoring engine bug (e.g., feature requests). Two annotators independently labeled these bug reports. During the labeling process, we filter bug reports according to the following criteria: (1) the issue should be related with refactoring engine, (2) the issue should contain input program, (3) the steps to reproduce bug should be well-illustrated, (4) the bug symptom should be clear (e.g., behavior change). Besides, we also included the bugs newly revealed in [10]. Following existing approaches [33], [34], [35], [36], [31], we measured the inter-rater agreement among the annotators via Cohen’s Kappa coefficient. Particularly, the Cohen’s Kappa coefficient was nearly 70% for the first 10% bug reports labeling results, thus we conducted a training session about labeling. After that, two annotators labeled 20% of bug reports (including the previous 10%), and Cohen’s Kappa coefficient reached 93%. After further discussion of the disagreements, Cohen’s Kappa coefficient was always more than 90% in subsequent labeling iterations (i.e., labeling 20% ~ 100% of bug reports with an interval of 10%). In each labeling iteration, two annotators discussed their disagreements until they reach a consensus. Finally, all bugs were labeled consistently. In total, we obtained 245 and 213 bug reports from ECLIPSE and INTELLIJ IDEA, respectively.

C. Seed Bug Reports Selection

The purpose of this step is to select the historical bug reports that contain high-quality seed input programs since they are vital for mutation-based testing [37], [38], [39], [40]. Specifically, to generate and grow a set of diverse input programs, the input programs should be complete and contain no syntax errors because it serves as the seed for subsequent mutations. Since it is too time-consuming and labor-intensive to validate input program in each bug report manually, so we

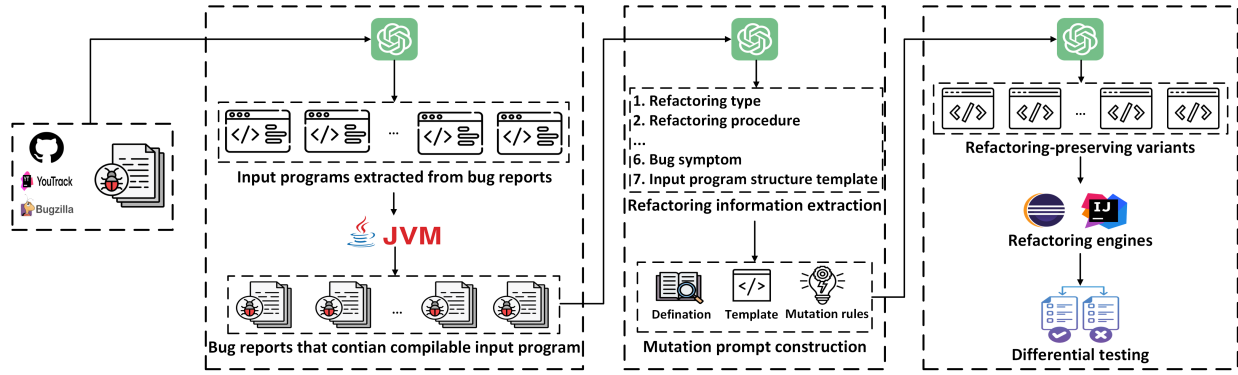


Fig. 4: Overall workflow of RETESTER.

adopt an automatic way by using a combination of few-shot-learning of LLM and JVM compiler. Different from traditional natural language corpus (e.g., news), a historical bug report (e.g., a bug report for Eclipse [27]) is a mixture of several elements such as input program, reproduction procedure (e.g., parameters for invoking a refactoring), environment description, and etc. Our goal is to extract input programs from the bug reports of refactoring engines. The input programs of bug reports are challenging to be extracted automatically because the input programs are usually embedded in a bug report within a webpage together with the natural language descriptions without any specific HTML tags or fields that make the relevant code snippets representing the input programs (e.g., the bug reports from Bugzilla are included as plain text comments [41]). Some input programs contain code comments with English texts, further make it difficult to distinguish them from other elements.

To tackle the aforementioned challenge, we leverage the power of LLM with few-shot learning [42], [43] from a small number of manual examples to extract bug-triggering input programs from the large amount of historical bug reports in our dataset. Specifically, the prompt to extract input program from bug report is in Table I, we manually constructed two examples in order to instruct the LLM. For each bug report, we create a new conversation thread in LLM during extraction. To construct the golden standard of the evaluation, we randomly sampled 100 historical bug reports from our dataset obtained through previous step, then we manually extracted input programs from those bug reports to serve as ground truth. To evaluate the effectiveness of our proposed method, we inspected the extraction results for those 100 historical bug reports against ground truth. In total, 98 out of 100 historical bug reports’ input programs are correctly extracted, indicating relatively high accuracy in the input program extraction. Only two input programs have been incorrectly extracted because the input programs are incomplete code snippets only contain few lines of code and deeply interleaved with the natural language descriptions. After extracting input programs from collected bug reports, we compiled them with Oracle JDK 22.0.1 using the “javac” command. As messages from com-

TABLE II: The seed bug reports information.

Source	Initial	Compilable	Mean LOC	Median LOC
ECLIPSE	245	101	11	9
INTELLIJ IDEA	213	66	11	9
Total	458	167	–	–

Initial = Number of initial bug reports, Compilable = Number of bug reports that contain compilable input program, Mean LOC = Mean of lines of code for the compilable input programs, Median LOC = Median of lines of code for the compilable input programs.

pilars are automatically generated, they follow strict formats. Upon completion of compilation, we parse the compilation messages to determine whether the JVM compiler accepts or rejects a test program. We filter out the bug reports whose input programs that are uncompileable. Finally, 291 out of 458 bug reports are filtered out, leaving 167 bug reports remained as seeds, the detailed information is in Table II. We calculated the mean and median lines of code for the compilable input programs derived from both ECLIPSE and INTELLIJ IDEA, finding that the mean and median values were both 11 and 9 lines, respectively.

D. Refactoring Information Extraction

For the seed historical bug reports obtained through the previous steps, we continue to extract the refactoring information from the bug reports by few-shot-learning. The purpose of this step is to get the necessary information (i.e., refactoring type and input program structure template) for the subsequent step. Meanwhile, by parsing the bug reports, we let LLM acts in a chain-of-thought way. Table III shows the prompt used to extract refactoring information from historical bug reports. During this step, we aim to get the input program structure template of the input program. This includes the critical program elements and structure related with the applied refactoring in the input program which are essential to trigger bug. 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. We adopt one-shot-learning for this step. Specifically, we manually construct one example, and then we incorporate the example into the prompt so as to instruct LLM to extract critical information

TABLE III: The prompt used to extract refactoring information from 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. Input programs;
3. Refactored programs;
4. Refactored program elements information (including element name, type, and positions.);
5. Refactoring procedures;
6. Bug symptoms;
7. Input program structure template.

The extracted information format should be **{Format}**.
The following is one example: **{Example}**

for the other bug reports. The middle of Figure 3 shows the extracted template for the top input program.

E. Mutation Strategy

For each template extracted above, we construct mutation strategy according to the bug-triggering input program characteristics obtained from previous study [10]. Input programs contain specific language features (e.g., lambda) or having complex class structures are more likely to trigger refactoring engine bugs and the top error-prone bug-triggering input program characteristics are related with Java language features (e.g., lambda expression, and Java generics), and complex class relationships (e.g., inner class, and anonymous class) [10], so, we choose three most error-prone input program characteristics as shown in Table IV in our study. Although there are diverse individual input program characteristic (38 types in [10]) that are error-prone, and different individual characteristic can also be combined to generate more complex input programs that could potentially trigger more bugs in the refactoring engine, we only select three of most error-prone characteristics to set a bound for the large search space of possible combinations.

Our prompt template used to perform the mutation is shown in Table V. Our prompt design is inspired by chain-of-thought (CoT) prompting, where instead of directly generating the final output, the prompt asks the model to finish task by breaking down the problem into sequential steps. Specifically, we first let LLM to understand current refactoring by giving the refactoring definition. Then, we give the LLM the input program structure template obtained from previous step and the bug-triggering input program characteristic in Table IV, and ask it to generate variants while preserving the original refactoring. Next, we add some extra requirements in our prompt template. For example, to reduce the uncompileable variants, we require the generated variants should conformance with specific JDK version. The variants are obtained through the Refactoring-preserving Transformation (RPT) as shown in Definition 1, which means the same refactoring should be applicable on both the original input program and its variant.

Definition 1 (Refactoring-Preserving Transformation (RPT)). Let P_1 be an input program and E_1 a program element within

TABLE IV: Input program characteristics applied for refactoring-preserving transformation.

Characteristic	Description
Lambda	Anonymous functions used to implement functional interfaces with a more streamlined syntax
Java generics	Java generics allow to create classes, interfaces, and methods that operate with unspecified types
Anonymous class	Class defined without a name, often used for one-time implementations of interfaces or abstract classes

TABLE V: The prompt template used to perform mutations.

Now, 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 according to the Input Program Structure Template, it is **{Template}**.
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 template. The variant format should be **{Format}**.

P_1 targeted for refactoring. Consider a refactoring operation O that is applicable to E_1 . A transformation function $Trans()$ defines a transformation such that $P_2 = Trans(P_1)$, where P_2 is the transformed program. This transformation is deemed refactoring-preserving if there exists a program element E_2 in P_2 and the refactoring operation O remains applicable to E_2 . Thus, the transformation $Trans()$ preserves the applicability of the refactoring operation from P_1 to P_2 .

Figure 3 bottom shows one variant generated by the LLM using the prompt template in Table V whose bug-triggering characteristic is set to Java anonymous class. This variant is refactoring-preserving, which means the “Pull Up” refactoring could still be applied on the “methodToBePulledUp()” method on variant. This variant successfully triggered one bug in the newest version of ECLIPSE (2024-09), resulting in an uncompileable refactored program. We have reported the bug together with the variant input program and bug reproduce steps to the ECLIPSE’s issue tracker systems [29]. We measure the LLM’s ability to perform RPT in RQ1.

F. Differential Testing

We manually refactor the variants in the newest version of refactoring engines by applying the same refactoring as in the seed bug reports, and compare the refactoring results of ECLIPSE and INTELLIJ IDEA using differential testing. To ensure the reliability and objectivity of the manual process, two annotators independently refactor each variant in the IDEs according to the reproduce procedures in the seed bug reports. Then, two annotators hold a meeting to resolve their disagreements. Based on prior work on testing refactoring engines [11], we adopt the following oracles:

TABLE VI: Seed historical bug reports used in our experiment.

ID	Source	Issue No.	Refactoring Type	Symptom
S-1	Eclipse	1533	Pull up	Compile error
S-2	Eclipse	1529	Inline method	Compile error
S-3	IDEA	142361	Extract variable	Compile error
S-4	IDEA	354116	Make static	Behavior change
S-5	IDEA	354122	Extract method	Compile error

Uncompilable Oracle. This checks if any of the refactored program contains syntax errors.

Warning Status Oracle. This compares and checks if the warning status from different refactoring engines are different. For example, one refactoring engine may produce a warning message but the other does not (this might occur due to the overly weak or strong preconditions checking in different refactoring engines).

Differential Oracle. This checks if (1) the refactoring has been performed successfully, (2) the refactored programs contain no syntax error, and (3) whether the refactored programs are the same.

As each oracle violation may indicate potential bugs in refactoring engines, we manually verify all violations. Before submitting the issues, we search in the corresponding bug-tracking systems of the target IDEs to avoid submitting duplicate reports. The search process involves looking for the keywords representing the refactoring type and bug symptoms. If any bug reports with similar input program and same symptom are found, we consider the bugs were already reported and will not report them.

IV. EVALUATION

A. Implementation

As shown in Table VI, we randomly select five historical bug reports from our seed dataset for our experiment. Specifically, two seeds are from ECLIPSE and three are obtained from INTELLIJ IDEA. These seeds cover five refactoring types and two symptoms. We use OpenAI API [44] to programmatically invoke the ChatGPT. The model type is gpt-4o-mini with the default settings [45] and the knowledge cut-off date is Oct 2023. For each seed, we iterate the characteristics listed in Table IV, and for each characteristic, we ask the LLM to generate ten variants. In conclusion, we generate 30 variants for each seed considering three different characteristics. In total, 150 variants are generated for the five seeds in Table VI. We set the JDK version in Table V to 22.0.1, and the corresponding refactoring definition is obtained from [46]. We manually tested the input program variants generated by our tool in the latest version of ECLIPSE (2024-09) and INTELLIJ IDEA (2024.2.4). All experiments are run on a workstation with 2.6GHz 6-core Intel Core i7 CPU and Windows 10, 64-bit operating system.

B. Research Questions

We investigate the following research questions in our experiments:

RQ1: *How effective is RETESTER in detecting refactoring engine bugs?*

RQ2: *How does RETESTER perform compared to other baseline?*

RQ3: *Which input program characteristics are more useful to reveal bugs?*

RQ4: *What is the contribution of input program structure template?*

V. EVALUATION RESULTS

A. RQ1: Effectiveness

Table VII presents detailed experimental results obtained using our tool, RETESTER. The “Refactoring” column lists the types of refactoring operations tested with our tool. It is important to note that the capabilities of our tool are not confined to the refactoring operations tested; its design is based on historical bug reports, facilitating easy extension to additional refactoring types by incorporating a diverse range of bug reports. Column “Template” tells about whether input program template is leveraged during mutation. The third column, “ET” indicates the time required to extract refactoring information for each type, which typically takes less than ten seconds. As discussed in IV-A, we generate 30 input program variants for each refactoring type. The “MT” column details the time taken to mutate these variants. Given that our tool leverages a large language model (LLM), it inherently generates variants that may contain syntax errors [47], [48]. However, the refactoring engine in ECLIPSE requires syntax-error-free input programs, otherwise refactoring operations cannot be performed, we subsequently filter the uncompileable variants using a JVM compiler. The “CV” column reports the number of compilable variants, with “Extract variable” having the highest count due to the simpler nature of its input programs, which facilitates easier mutation. Column “RPV” shows the number of refactoring-preserving variants, while most of the compilable variants preserve the original refactoring, there exists variants for “Make static” refactoring that are not refactoring-preserved. This happens because the method to be refactored in the input program is mutated to a static method in variants, however, one of the preconditions to perform “Make static” refactoring is that the method to be refactored should not already been static [49], thus leading to eight variants cannot applying “Make static” refactoring. The “Oracles” column calculates the number of variants for each oracle as outlined in our methodology. Predominantly, most bugs are identified by our “Uncompilable Oracle,” indicating that the refactoring engine often produces refactored programs with syntax errors. This aligns with previous studies on refactoring engine bugs where compilation errors are the most common symptom [10]. For the same input program, ECLIPSE and INTELLIJ IDEA may yield different syntax-error-free refactored programs due to the different default refactoring configurations employed by each engine. For example, when performing “Make static” refactoring for a method having parameters, INTELLIJ IDEA would declare the parameters as “final” during refactoring while ECLIPSE would not. We

TABLE VII: Experimental result of RETESTER.

Refactoring	Template	ET (s)	TGV	MT (s)	CV	RPV	Oracles			Bugs	
							UC	WS	Diff.	EC	IDEA
Extract method	Y	6	30	87	27	27	1	0	0	1	0
	N	7	30	131	28	28	0	0	0	0	0
Inline method	Y	8	30	91	26	26	5	0	0	5	0
	N	7	30	81	20	20	3	0	0	3	0
Extract variable	Y	6	30	78	30	30	0	0	0	0	0
	N	6	30	73	25	25	0	0	0	0	0
Pull up	Y	7	30	105	20	20	8	1	0	7	2
	N	8	30	109	20	20	2	0	0	1	1
Make static	Y	10	30	104	25	22	0	0	2	0	0
	N	11	30	179	26	21	1	0	0	1	0
Average	–	7.6	30	103.8	25	24	–			–	
Total	–	76	300	1038	247	239	20	1	2	18 (15)	3

Template = Whether input program template is used during mutation, ET = Time taken in seconds to extract refactoring information, TGV = Total generated variants, MT = Mutation time for TGV in seconds, CV = Compilable variants, RPV = Refactoring-preserving variants; Oracles: UC = Uncompilable Oracle, WS = Warning Status Oracle, Diff. = Differential Oracle; EC = ECLIPSE, IDEA = INTELLIJ IDEA.

manually analyzed the differential results for “Make static” refactoring and confirmed that all two variants were correctly refactored. Those two differential results were produced because the default configurations for “Make static” refactoring are slightly different in those two refactoring engines as described above, thus leading to two false positive. However, the default configuration for other refactoring operations remain the same. According to previous study [10], most of the bugs (97%) could be triggered by the default initial configuration of refactoring engines, and the default configuration for different refactoring engine could be changed to the same by setting up the same configuration parameters. The last two columns record the number of bugs we identified in ECLIPSE and INTELLIJ IDEA, which is 18 and three, respectively. For ECLIPSE, there are three overlap bugs when testing “Inline method” refactoring with and without input program template during mutation stage, thus leading to 15 unique new bugs. Prior to report, we searched the refactoring engine’s bug-tracking system to avoid duplicates. Notably, “Pull up” refactoring accounts for 11 out of the 18 reported bugs, primarily due to its involvement with complex class relationships that complicate the refactoring process.

Table VIII presents comprehensive details of the 18 bug reports we submitted, including 15 bugs identified in RQ1 and three discovered in RQ4. The “IDE” column identifies the refactoring tool under evaluation. The columns “Issue No.” and “Refactoring Type” specify the respective issue numbers in the bug-tracking systems and their corresponding refactoring types. The “Symptom” column provides a summary of the symptoms associated with each bug, while the final column reports the current status of each issue. As of the submission of this paper, seven have been confirmed by developers. Notably, “Pull Up” and “Inline Method” refactoring revealed the highest number of bugs, with eleven and five instances respectively, predominantly resulting in compile errors. This observation supports findings from existing studies [10], which indicate that compile errors are the most prevalent symptom in refactoring engine bugs. Furthermore, four out of the eigh-

TABLE VIII: Detailed information for our submitted bug reports.

ID	IDE	Issue No.	Refactoring Type	Symptom	Status
B-1	Eclipse	1785	Extract Method	Compile error	Submitted
B-2	Eclipse	1824	Make Static	Compile error	Confirmed
B-3	Eclipse	1783	Inline Method	Compile error	Submitted
B-4	Eclipse	1781	Inline Method	Compile error	Submitted
B-5	Eclipse	1780	Inline Method	Compile error	Fixed
B-6	Eclipse	1779	Inline Method	Compile error	Submitted
B-7	Eclipse	1778	Inline Method	Compile error	Submitted
B-8	Eclipse	1777	Pull Up	Compile error	Submitted
B-9	Eclipse	1776	Pull Up	Compile error	Submitted
B-10	Eclipse	1775	Pull Up	Compile error	Submitted
B-11	Eclipse	1774	Pull Up	Failed refactoring	Submitted
B-12	Eclipse	1773	Pull Up	Compile error	Fixed
B-13	Eclipse	1772	Pull Up	Compile error	Submitted
B-14	Eclipse	1766	Pull Up	Compile error	Submitted
B-15	Eclipse	1823	Pull Up	Compile error	Fixed
B-16	IDEA	364110	Pull Members Up	Compile error	Confirmed
B-17	IDEA	362805	Pull Members Up	Compile error	Confirmed
B-18	IDEA	362804	Pull Members Up	Compile error	Confirmed

The issues of INTELLIJ IDEA, and ECLIPSE can be found at <https://youtrack.jetbrains.com/issue/IDEA-XXX>, and <https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/XXX>, where “XXX” can be replaced with the concrete numbers in **Issue No.**

teen bugs have been officially confirmed by their respective developers, highlighting the ongoing engagement between our research efforts and the development community to enhance the reliability of refactoring tools.

B. RQ2: Comparison with Baseline

We selected the state-of-the-art Gligoric et al. [18] testing approach as our baseline, which perform systematic testing of refactoring engines by applying refactoring operations at randomly chosen program elements within real-world projects. To ensure a fair comparison, we replicate this process by employing a large language model (LLM) to propose refactoring for each input program derived from the seed bug reports. Specifically, for each program, the LLM is instructed to suggest ten possible refactoring operations, mirroring the number of transformations we have predetermined for each category, such as “Lambda”. Subsequently, these suggested refactoring

TABLE IX: Different transformations in our tool compared to the baseline.

Transformation	ET (s)	TGV	MT (s)	CV	RPV	Bugs	
						EC	IDEA
RETESTER _L	8	50	146	46	45	4	2
RETESTER _G	8	50	91	39	37	3	0
RETESTER _A	9	50	131	43	43	6	0
Gligoric et al. [18]	9	50	152	45	45	0	0

ET = Extract refactoring information time, time is in seconds; TGV = Total generated variants, MT = Mutation time for TGV in seconds, CV = Compilable variants, RPV = Refactoring-preserving variants, EC = ECLIPSE, IDEA = INTELLIJ IDEA.

operations are implemented in ECLIPSE and INTELLIJ IDEA to uncover potential bugs.

Table IX provides comparative results for each component of our tool versus the baseline. In the notation used, RETESTER_L denotes the application of only the Lambda transformation, as detailed in Table IV. Similarly, RETESTER_G and RETESTER_A indicate exclusive application of Java generics and anonymous class transformations, respectively, within our tool. Based on the five seeds in Table VI, we generate ten variants for each seed by implementing each transformation, resulting in a total of 50 variants. The concluding two columns display the number of bugs identified in ECLIPSE and INTELLIJ IDEA by each component of our tool and the baseline, respectively. Notably, our tool successfully detected 15 new bugs. Without performing mutation on the input program, Gligoric’s approach fails to find any bugs using the same set of seed input programs as our approach. This substantial discrepancy highlights the enhanced efficacy of our tool in identifying potential issues within refactoring engines.

C. RQ3: Contribution of Different Characteristics

Table IX shows the number of bugs identified by applying various input program characteristics through our tool. Specifically, RETESTER_L, RETESTER_G, and RETESTER_A represent configurations where only Lambda transformations, Java generics, and anonymous class transformations are applied, respectively. The results show that lambda and anonymous class transformations each exposed six bugs, whereas Java generics revealed three bugs. In terms of mutation time, lambda transformations are the most time-consuming ones, taking 146 seconds. Java generics generated the least number of compilable variants, with only 39 out of 50, largely due to the complex syntax associated with generics which makes it challenging to generate syntactically correct variants. The majority of bugs, 13 out of 15, were detected in ECLIPSE, with anonymous class transformations revealing six, lambda transformations revealing four, and Java generics uncovering three. This distribution suggests that developers of ECLIPSE should particularly focus on improving refactoring operations that involve complex class relationships, type inference, and code transformations. In future, we plan to consider further enhancements in the bug detection capability by combining individual characteristics or by generating variants that in-

corporate a broader range of diverse and bug-triggering input program characteristics [10].

D. RQ4: Effectiveness of the Template

We conducted an ablation study to elucidate the specific contributions of the input program template to the mutation process. For this study, we modified the mutation prompts in Table V by substituting the template information with details from the actual input program. Specifically, we revised the prompt “You need to generate the variant according to the Input Program Structure Template, it is {Template}.” to “You need to generate the variant according to the Input Program, it is {Input Program}.” Additionally, the instruction “Please generate one edge case variant considering different edge usage scenarios of {Characteristic} based on the template.” was altered to “Please generate one edge case variant considering different edge usage scenarios of {Characteristic} based on the Input Program.” The input programs utilized for this process were derived from our refactoring information extraction step. Except for these modifications to the mutation prompts, all other experimental settings remained unchanged.

Table VII presents the outcomes of the experiment conducted without utilizing the input program template in the “Template” column marked as “N”. The 1st two columns indicate the number of bugs revealed: three for the “Inline Method” refactoring, two for the “Pull Up” refactoring, and one for the “Make Static” refactoring. In comparison to the original results listed in Table VII, where 15 bugs were detected, only six bugs were identified when the template was not used. This reduction in bug detection can be attributed to the constraints imposed by using specific input programs, which significantly narrows the LLM’s search space for generating variants. For instance, in the template shown in Figure 3 middle, the code within the method methodToBePulledUp() is abstracted to suggest potential logic interacting with the outer class context. Conversely, in the input program depicted at the top of the figure, the actual logic within the method is retained, restricting the LLM to perform further mutations based on this specific method logic, thereby resulting in less variant diversity. We also observe that the approach without prompt template can only detect bugs with uncompileable oracle (UC). Without getting guidance on the abstracted program structures, the approach without template can only rely on knowledge about the detailed code logic and Java syntax to generate variants that result in syntactically incorrect programs, which limits its capability in detecting bugs that require more complex oracles (i.e., Warning Status Oracle and Differential Oracle). We conducted further analysis on the bugs revealed with and without the template to assess overlap and uniqueness. The Venn diagram in Figure 5 shows that three bugs are overlapped, constituting 50% of the bugs detected by RETESTER without using the template. Additionally, 12 and three unique bugs were identified exclusively with and without the template, respectively, illustrating the significant role of the input program template in facilitating diverse code mutations.

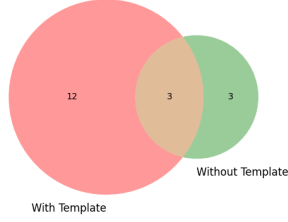


Fig. 5: The venn diagram for the number of bugs detected by RETESTER with and without input program template.

VI. IMPLICATION

Based on our study and analysis, we discuss the implications for researchers and refactoring engine’s developers.

Implication for Researchers. Our study and automated testing framework lay the foundation for future research in three promising directions. First, our seed dataset serve as cornerstone for future research in automated refactoring engine testing. The lines of code (LOC) analysis result in Table II indicates that bug-revealing input program is usually having a small size, with mean and median LOC is 11 and 9, respectively. *This observation can be leveraged for improving the effectiveness of test generation for refactoring engine validation. A test generation tool should focus on generating small but complex input programs rather than big but simple ones.* Second, bug-triggering input program characteristics are critical for refactoring engine testing. Input programs own specific characteristics (e.g., anonymous class) are more error-prone. This finding indicates that when generating test program to reveal bugs, we should take the program characteristics into account. *Automated testing techniques in the area of compiler, static analyzer, and others which take program as input could all benefit from incorporating the error-prone input program characteristics.* Our study serves as a preliminary study to motivate future research on using a richer set of transformations based on the bug-triggering input program characteristics for improving the reliability of static analysis tools. *Third, the input program structure template obtained through our tool could be used for template-based techniques as their references for designing templates, thus improving effectiveness.*

Implication for Developers. Our study identifies new bugs in both ECLIPSE and INTELLIJ IDEA leveraging the historical bug reports and error-prone input program characteristics. *Developers of refactoring engine should consider adding a checklist for those characteristics when testing their refactoring operations in IDEs.* This could be obtained by analyzing the historical bug reports. Since manually writing test input programs for each refactoring could be labor-intensive and time-consuming, *refactoring engine developers could leverage the LLM as assistant while testing considering its effectiveness.* Developers should set a higher priority for input programs that

contain certain characteristics (e.g., Lambda), since they tend to have a higher bug-revealing capability.

VII. THREATS TO VALIDITY

We identify the following threats to the validity:

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 referred to the previous studies [31], [50], and then adopted an open-coding scheme. During the labeling process, two annotators independently labeled bugs, any disagreement was discussed at a meeting until a consensus was reached. As our dataset and our approach only focus on Java input programs generation, the findings may not generalize for other programming languages beyond Java.

External. The external threat to validity mainly lies in the dataset used in our study. To reduce this threat, we systematically collected refactoring engine bugs as presented in Section III. To ensure the diversity and generalization of the considered refactoring engines, we choose two of the most popular refactoring engines (i.e., ECLIPSE and INTELLIJ IDEA) as our studied target.

VIII. RELATED WORK

LLM for testing. Recent advancements in Large Language Models (LLMs) have significantly boost software testing methodologies. TitanFuzz introduces LLMs to fuzz deep learning libraries by generating seed inputs specified through API-related prompts and employing mutation strategies like code masking, followed by LLM completion [51]. FuzzGPT extends this by using few-shot prompts, providing examples of code and descriptions to generate test cases for deep learning libraries [43]. Similarly, LAST leverages LLMs to validate SMT solvers by generating diverse formulas [52]. WhiteFox marks a significant advancement as the first white-box compiler fuzzer that uses source-code information with LLMs to test compiler optimizations, uncovering deep logic bugs in deep learning compilers [53]. We are the first to leverage LLM for refactoring engine testing by combing historical bug reports and error-prone input program characteristics.

IX. CONCLUSION

In this paper, we introduced a novel approach to test refactoring engine based on historical bug reports and bug-triggering input program characteristics empowered by Large Language Model (LLM). RETESTER does not require manually designing templates and can easily generalize to diverse refactoring types. The experimental results show that RETESTER can detect various new bugs in both ECLIPSE and INTELLIJ IDEA. In total, we identified 18 new bugs in the latest version of both refactoring engines. By the submission time of our paper, seven bugs were confirmed, three of them were fixed.

Data Availability. The data is available at [25].

ACKNOWLEDGMENTS

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

REFERENCES

- [1] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [2] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th working conference on reverse engineering*. IEEE, 2004, pp. 144–151.
- [3] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.
- [4] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," *Information and Software Technology*, vol. 93, pp. 186–199, 2018.
- [5] M. Wahler, U. Drofenik, and W. Snipes, "Improving code maintainability: A case study on the impact of refactoring," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 493–501.
- [6] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.
- [7] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, 2018.
- [8] (2024) Eclipse. [Online]. Available: <http://www.eclipse.org/>
- [9] (2024) IntelliJ idea. [Online]. Available: <http://www.jetbrains.com/idea/>
- [10] H. Wang, Z. Xu, H. Zhang, N. Tsantalis, and S. H. Tan, "An empirical study of refactoring engine bugs," *arXiv preprint arXiv:2409.14610*, 2024.
- [11] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "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*, 2007, pp. 185–194.
- [12] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, and M. Cornélio, "Saferefactor-tool for checking refactoring safety," *Tools Session at SBES*, pp. 49–54, 2009.
- [13] M. Mongiovi, "Scaling testing of refactoring engines," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 15–17.
- [14] G. Soares, "Making program refactoring safer," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2*, 2010, pp. 521–522.
- [15] G. S. Soares, "Automated behavioral testing of refactoring engines," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 49–52.
- [16] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying overly strong conditions in refactoring implementations," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 173–182.
- [17] G. Soares, R. Gheyi, T. Massoni, M. Cornélio, and D. Cavalcanti, "Generating unit tests for checking refactoring safety," in *Brazilian Symposium on Programming Languages*, vol. 1175, 2009, pp. 159–172.
- [18] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "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, 2013, pp. 629–653.
- [19] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 151–160.
- [20] J. Choi, G. An, and S. Yoo, "Iterative refactoring of real-world open-source programs with large language models," in *International Symposium on Search Based Software Engineering*. Springer, 2024, pp. 49–55.
- [21] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 202–206.
- [22] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 582–586.
- [23] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [24] Z. Zang, F.-Y. Yu, A. Thimmaiah, A. Shi, and M. Gligoric, "Java jit testing with template extraction," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1129–1151, 2024.
- [25] (2024) To be open-sourced.
- [26] E. Lacker, J. Kim, A. Kumar, L. Chandrashekar, S. Paramaiahgari, and J. Howard, "Statistical analysis of refactoring bug reports in eclipse bugzilla," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 9–13.
- [27] (2024) Pull up method refactoring for method in the inner class fails. [Online]. Available: <https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1533>
- [28] (2024) Add outer class checking to pull up refactoring. [Online]. Available: <https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/1590>
- [29] (2024) [bug][pull up refactoring] pull up refactoring for the method in anonymous class produce uncompileable code. [Online]. Available: <https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/1766>
- [30] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 294–305.
- [31] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "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*, 2021, pp. 968–980.
- [32] (2022) Github apis. [Online]. Available: <https://docs.github.com/en/rest?apiVersion=2022-11-28>
- [33] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 385–396.
- [34] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "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*, 2019, pp. 510–520.
- [35] J. Wang, G. Xiao, S. Zhang, H. Lei, Y. Liu, and Y. Sui, "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*, 2023, pp. 476–488.
- [36] H. M. Win, H. Wang, and S. H. Tan, "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*, 2023, pp. 644–656.
- [37] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 700–711.
- [38] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.
- [39] T. Gao, J. Chen, D. Wang, Y. Guo, Y. Zhao, and Z. Wang, "Selecting initial seeds for better jvm fuzzing," *arXiv preprint arXiv:2408.08515*, 2024.
- [40] S. Li, T. Theodoridis, and Z. Su, "Boosting compiler testing by injecting real-world code," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 223–245, 2024.

- [41] (2024) Bug 92519 - [refactoring] inline method - result does not compile. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=92519
- [42] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.
- [43] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [44] (2024) Openai api. [Online]. Available: <https://platform.openai.com/docs/overview>
- [45] (2024) Openai api default settings. [Online]. Available: <https://platform.openai.com/docs/api-reference/chat/create>
- [46] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [47] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.
- [48] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.
- [49] (2023) Make static refactoring preconditions. [Online]. Available: <https://github.com/eclipse-jdt/eclipse.jdt.ui/issues/590>
- [50] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.
- [51] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.
- [52] M. Sun, Y. Yang, Y. Wang, M. Wen, H. Jia, and Y. Zhou, "Smt solver validation empowered by large pre-trained language models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1288–1300.
- [53] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.