Using LLM for Code Refactoring: an Evaluation of Effectiveness and Correctness

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
email:luigi.lavazza@uninsubria.it

Gabriele Rotoloni
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
email:gabriele.rotoloni@uninsubria.it

***Abstract*—Code smells are indicators of poor software structure and organization, suggesting potential maintenance challenges and future issues. To enhance code quality and eliminate these smells, developers often engage in code refactoring, which involves restructuring existing code without altering its external behavior. This process, while essential, can be time-consuming and may not require advanced programming skills. Recent advancements in large language models (LLMs) have prompted investigations into their ability to automate code refactoring. This paper presents an experimental evaluation of LLMs' capabilities in refactoring code within development environments that utilize test cases for code units. Recognizing the risk of regression—where changes inadvertently disrupt previously functioning features—developers typically establish a comprehensive set of test cases for each software unit. Our approach leverages automatic code smell detection tools alongside existing test cases to streamline the refactoring process, maximizing automation while maintaining code integrity. The findings of this study contribute to the objectives of the conference by addressing the impact of AI on software development; specifically, the paper contributes to the exploration of LLMs' potential to enhance code quality through automated refactoring techniques.**

Keywords–Code smells; code refactoring; LLM for software development

I. Introduction

Code smells are characteristics of software code, usually connected with poor code structure and organization, that indicate that several types of problems are likely to occur. Among these problems are the difficulty to maintain the code and the proneness to future issues. To improve the quality of the code and eliminate code smells, the code has to be "refactored" [1]. Code refactoring is the process of restructuring existing software code without changing its external behavior. In this respect, it is important to stress that code smells do not indicate defects in the code that cause incorrect behavior: therefore, while taking care of code smells, via code refactoring, we have to maintain the behavior of code unaltered.

Now, code refactoring is a time-consuming activity, which often does not require great programming skills. Therefore, the possibility of automatically refactoring code, especially using LLMs, is widely investigated.

The work reported here was partly supported by Fondo per la Ricerca di Ateneo, Università degli Studi dell'Insubria.

In this paper, we report about the experimental evaluation of LLMs' code refactoring capabilities in development environments where test cases for code units are available.

It is well known that whenever the code is changed, regression may occur. That is, a change in the codebase inadvertently causes previously functioning features or functionalities to fail or behave incorrectly. Because of the possibility of regression, well organized software developers define a set of test cases for each software unit. These test cases are useful not only to make sure (as far as possible) that the initially released code is correct, but are used after any change involving the unit, to make sure that the change preserved the behavioral properties of code.

We exploit both automatic code smell detectors and the available test cases to automatize as much as possible the process involving code refactoring performed by LLMs.

The paper is organized as follows. Section II provides some background: the concepts of code smell and code refactoring are introduced; the previous work concerning the usage of LLMs for code refactoring is briefly reviewed. Section III presents the objectives of the paper, highlighting the benefits that can derive from the proposed usage of LLMs. Section IV describes the empirical study we carried out to evaluate the effectiveness and correctness of code refactored by LLMs. The results of the study are described in section Section V. In Section VI, we discuss the results of the study and draw conclusions and lessons learned; we also outline some directions for further work.

II. Background

In this section, we recall the concepts of code smell and code refactoring, and we briefly report about the existing literature on using LLMs for code refactoring.

A. Code smells

The term "code smells" was first introduced by Kent Beck and Martin Fowler in 1999 [1]. Code smells refer to certain patterns or characteristics in code that indicate deeper problems within the software design or implementation. These are not outright bugs but rather signs that the code may be poorly structured, difficult to maintain, or prone to future issues. Specifically, code smells are described as indicators of potential issues in the code that may require refactoring. The idea behind code smells is to help developers recognize patterns that could lead to problems, thereby encouraging them to take action to improve the code quality.

Addressing code smells is essential for maintaining a healthy codebase. By refactoring code to eliminate these smells, developers can improve readability, enhance maintainability, and reduce the likelihood of introducing new bugs. This proactive approach not only streamlines the development process but also fosters a more collaborative environment, as cleaner code is easier for team members to comprehend and work with. Furthermore, addressing code smells can lead to better performance and scalability, ultimately resulting in a more robust software product.

The benefits of eliminating code smells extend beyond immediate code quality improvements. A cleaner codebase can significantly reduce the time and effort required for future enhancements and bug fixes, leading to increased productivity and lower costs in the long run. Additionally, it can enhance the overall developer experience, as working with well-structured code can boost morale and encourage best practices within the team. In summary, recognizing and addressing code smells is a vital practice in software development that contributes to the longevity and success of a project.

Of the many types of code smells that have been described in the literature, we concentrate on the following ones, which are among the most dangerous and the best supported by tools:

- **Feature Envy** occurs when a method in one class is more interested in the data of another class than its own. This often indicates that the method should be moved to the class it is envious of, as it suggests a poor separation of concerns and can lead to tight coupling between classes.
- **Type Checking** refers to the practice of using conditional statements to check the type of an object before performing operations on it. This smell often indicates that polymorphism is not being utilized effectively, leading to code that is less flexible and harder to maintain. It suggests that the design may need to be reconsidered to leverage object-oriented principles.
- A **Long Method** is a method that is doing too much and should be broken down into smaller, more focused methods. Shorter methods enhance readability and make it easier to test and debug code.
- A **God Class** is a class that has too many responsibilities, often containing a large amount of data and many methods. This smell violates the Single Responsibility Principle and can lead to a codebase that is difficult to manage and understand. Refactoring such classes into smaller, more cohesive classes can improve maintainability.

B. Code refactoring

Code refactoring is the process of restructuring existing computer code without altering its external behavior or functionality. The primary goal of refactoring is to improve the internal structure of the code, making it cleaner, more efficient, and easier to understand. This practice involves making small, incremental changes that enhance the code's readability and maintainability while ensuring that the original functionality remains intact. By focusing on improving the design and organization of the code, developers can reduce technical debt and facilitate future enhancements or modifications.

During the refactoring process, developers may rename variables, extract methods, or reorganize classes, among other techniques. However, it is crucial that these changes do not introduce new features or alter the existing behavior of the software. Maintaining the same functionality ensures that the refactored code continues to meet the original requirements and passes all existing tests. This careful approach allows teams to enhance the quality of their codebase while minimizing the risk of introducing bugs or regressions, ultimately leading to a more robust and maintainable software product.

C. Previous work on using LLM for code refactoring

Recently, code refactoring via (or supported by) LLMs has been the object of a huge amount of research. Here we just mention the very recent research that has highlighted the need to get a reliable feedback on the quality of the code produced by LLMs.

Liu et al. [2] applied LLMs to identify refactoring opportunities and then perform the refactoring, using a dataset of 180 real-world refactorings from 20 projects. They found that "*13 out of the 176 solutions suggested by ChatGPT and 9 out of the 137 solutions suggested by Gemini were unsafe in that they either changed the functionality of the source code or introduced syntax errors*."

Cordeiro et al. [3] discussed the current limitations of LLMs in refactoring. Noticeably, they suggest that "to address the limitations of LLM-based refactoring hallucinations, future research can focus on generating comprehensive test cases."

Both the results by Liu et al. and the suggestions by Cordeiro et al. support our idea that the code refactored by LLMs should be tested for regression, and possibly sent back to refactoring.

III. The goals of the paper

As mentioned in Section II-C above, several studies have addressed the automatic refactoring of code via LLMs. In this paper, we consider a specific kind of situation that occurs quite often in software development environments characterized by a high level of automation. Specifically, we consider those situations when

1) The existence of code smells is detected via automated tools.
2) Code is provided with test cases, so that it is possible to rerun the test cases every time the code is updated, to ensure that existing functionality has not been altered.

Accordingly, the process we want to achieve (illustrated in Figure 1) is organized as follows: When a code smell is detected, the code and the type of smell are fed to LLM, together with the request to refactor the code so that the smell is removed, while the code functionality is preserved. The code provided by the LLM is checked via the automated

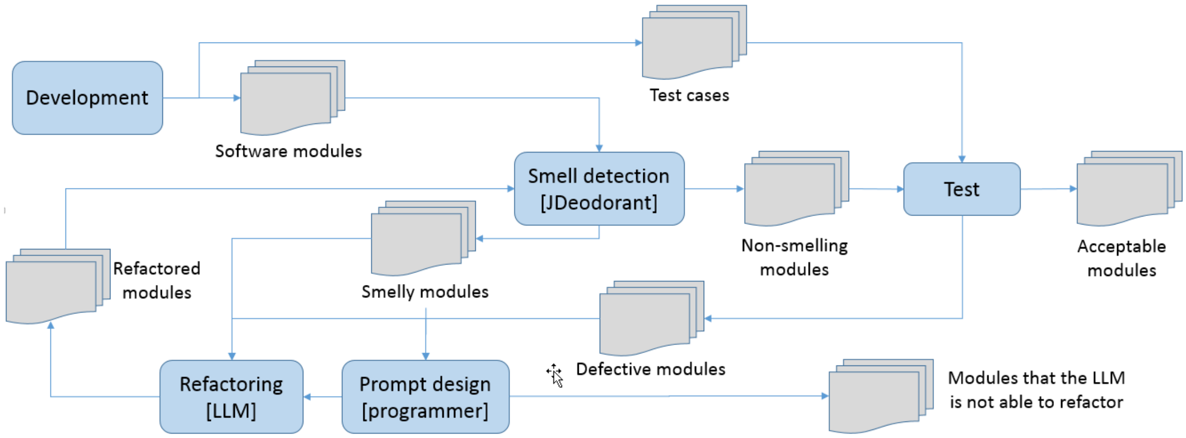

Fig. 1. The smell detection and elimination process.

code smell detection tool, to verify that the smell has actually been removed (or at least, it has been substantially reduced), then the test cases are run, to verify that functionality was not damaged. If all the checks succeed, the code provided by the LLM can be safely adopted. Otherwise, a new prompt is devised, to help the LLM achieve a better refactoring. If no acceptable code is obtained after a few cycles, the programmer can decide to give up on using the LLM and proceed to refactor the smelly code manually.

The process we pursue aims to automating the most trivial operations and leaving to developers the activities that require an expert and smart evaluation of code.

IV. The Study

To carry out the study, we needed some code to refactor. This code had to also be equipped with test cases. To this end, we looked for source code on GitHub.

To detect automatically the existence of smells in the code, we looked for suitable tools. The available ones are mostly usable with Java code, hence we limited the search of code to programs written in Java.

Of the available tools, the one that suits best our requirements is JDeodorant [4], [5]. It is integrated in Eclipse [6], arguably the most popular open-source development IDE for Java. JDeodorant supports the detection of a limited number of code smells, with respect to the catalog of code smells described by Fowler [1]. Namely, JDeodorant detects the Feature Envy, Type Checking, Long Method, and God Class smells. This set of code smells is large enough for our purposes. In fact, the supported smells are also among the most frequent, hence the ones that most likely developers have to deal with.

We found the following open-source code, which has the required characteristics:

– Karate [7], a tool for automated testing that combines API test-automation, mocks, performance-testing, and UI automation.
– JUnit4 [8], a testing framework for Java.
– Spring Cloud Gateway [9], an API gateway built on Spring Framework and Spring Boot.

V. Execution of the Study and Results

We first performed all test cases to ensure that there were no problems with the code: all tests were successfully passed. Then, we scanned the projects' code using JDeodorant to identify the code smells that affected the code: we collected a total of 30 smells, as shown in Table I.

TABLE I
Smells found per type and project.

	Karate	JUnit4	Spring	Total
Type Checking	1	3		4
Feature Envy	3	11		14
God Classes	5			5
Long Methods	1		6	7

To refactor the code, we used ChatGPT 4o-mini [10]. Our intent was to simulate how a casual developer would normally use LLMs to improve code: typically chatting with the free version of ChatGPT. Therefore, we chose to use zero-shot prompts and, when the LLM was not able to refactor the code from the initial prompt alone, we provided follow-up prompts with additional context, such as test cases, related code snippets, and details about the sections flagged by JDeodorant.

The initial prompt for each code smell is in Listing 1.

Listing 1. Initial prompt for code smell refactoring.

```
Our code smell analysis tool detected a code
    smell of type [CODE SMELL TYPE] in the
    following [code/method]:
[CODE]
Refactor the code so that the detected code
    smell is removed.
```

When the LLM was not able to refactor the code with just the first prompt, subsequent prompts were used to provide additional information and context—like test cases or additional pieces of code—to the model. We stopped prompting the LLM either when the smell was successfully removed, or when we perceived that the LLM was going in circles and any subsequent iteration would have been useless. According to the process described in Figure 1, a code smell was considered removed when JDeodorant did not detect the smell in the refactored code and the tests were successfully passed. This condition does not guarantee that the smell had been refactored in a correct and satisfactory way, but simulates what developers would do: if they trust JDeodorant to detect smells, they probably trust it also when it does not detect any smell.

For evaluating the quality of the refactoring performed by the LLM, we used the automatic refactoring functionality of JDeodorant to get a refactored code baseline.

A. Refactoring Feature Envy

As mentioned in Section II-A, a Feature Envy smell occurs when a method `mA` of class `A` uses some features `X` from a different class `B`. We define `mA` as the "envious" method, and `X` as the object of the envy. The preferable situation is that `B` provides some method `mB` that uses `X`, so that `mA` can call `mB` instead of violating encapsulation via direct access to `X`.

In 7 cases out of 14, ChatGPT refactored the smell by transferring the envious code from `mA` to `B`. In two cases, ChatGPT generated a sub-class of `B` and put the envious code in it.

In some cases, ChatGPT generated an improved, more readable version of the original code, as shown in Listing 2. This behavior is not achievable by JDeodorant, as it refactors code smells systematically, without "looking" at the code, context, etc.

Listing 2. Feature Envy smell has been moved in the `Response` class. The code inside the method has been rewritten.

```
// Code in the original method
return res.getStatus() != 200 &&
      !res.json().<String>get("value")
       .contains("unexpected alert open");

// Code generated by ChatGPT
if (this.getStatus() == 200) {
            return false;
}
String value;
try {
      value = this.json().get("value");
} catch (Exception e) {
      return true;
}
return value == null || !value.contains("unexpected
      alert open");
```

In six cases, ChatGPT created a new class U that contains a method mU identical to mA. Clearly, mU is as envious as mA was, but the smell was not detected by JDeodorant in four of these six cases, because mU was defined as static, and JDeodorant does not recognize Feature Envy smells in static methods [11]. In four cases, the solution given by ChatGPT was proposed after the initial prompt.

In a single case, ChatGPT introduced two new methods m1A and m2A, such that mA calls m1A, which calls m2A: in this way, ChatGPT was able to remove the definition of variables that led JDeodorant to recognize the smell. This case is shown in Listing 3: since the sub-methods take a `Description` object as a parameter instead of relying on the `description = runner.getDescription()` variable, JDeodorant is not able to detect the smell anymore.

Listing 3. ChatGPT solves the smell incorrectly and JDeodorant still detects the smell.

```
// Smelly code
Description description = runner.getDescription();
... // rest of the code

// Cleaned code by ChatGPT
applyOrderingIfPresent(runner.getDescription(),
     runner);

// applyOrderingIfPresent contains the code of the
     original method and a second sub-method that
     contains one statement of the original method.
```

B. Refactoring Type Checking

Type Checking occurs when the type of an object is explicitly checked before calling a method, instead of using polymorphism. In Java, the type of an object is usually obtained via the `instanceof` operator for class types, and the == operator for enum types.

For all of the 4 smell instances, ChatGPT refactored the code so that JDeodorant did not detect the smell anymore, but the refactored code was definitely not satisfactory.

In two cases, the == operator was simply replaced with the `equals` method, as in Listing 4. JDeodorant does not detect the `getClass().equals(ClassName.class)` condition as a Type Checking smell; JDeodorant would detected the small if the code was `getClass() == ClassName.class` [12]. In one instance, ChatGPT removed the reference to the enum completely, by replacing `type == ValueType.MAP` with `value instanceof Map<?, ?>)`. In this case, JDeodorant does not detect the smell anymore because it only flags code it can refactor: since it cannot apply polymorphism to a base Java class (`Map`), it does not detect the smell.

Listing 4. Type Checking solved by ChatGPT using `equals`.

```
// Smelly condition of the code for the class
e instanceof AssumptionViolatedException

// Condition generated by ChatGPT
e.getClass()
      .equals(AssumptionViolatedException.class)
```

In one case, ChatGPT generated code that uses conditional method invocation (see Listing 5). The "solution," which still performs type checking (though in a different way), is potentially dangerous, because it changes the semantics of the code. In fact, the original code prescribes that any sub-class of `Test` that is not a `TestCase` execute `toString()`. Instead, the refactored code executes `getName()` whenever present. If `getName()` is implemented in a class different from `TestCase` with a different behavior, unexpected results are possible.

Listing 5. Type Checking solved via conditional method invocation.

```
// Smelly method
private String getName(Test test) {
      if (test instanceof TestCase) {
            return ((TestCase) test).getName();
      } else {
            return test.toString();
      }
}
```

```
// Cleaned code generated by ChatGPT
public String resolveName(Test test) {
      try {
            Method getNameMethod =
                  test.getClass().getMethod("getName");
            Object name = getNameMethod.invoke(test);
            if (name instanceof String) {
                  return (String) name;
            }
      } catch (NoSuchMethodException |
          IllegalAccessException |
          InvocationTargetException ignored) {}
      return test.toString();
}
```

C. Refactoring God Classes

A class is a God Class if it takes on multiple responsibilities. When this happens, portions of the code (attributes and methods) happen to be very cohesive and rather disconnected from the rest of the class. The preferable situation is that these portions of code are themselves implemented as classes, so that each class has different responsibilities.

ChatGPT was able to refactor three out of five God Classes; in two of these three cases, the solution was satisfactory, being obtained by identifying the aforementioned responsibility-related portions of code and moving them in new classes. Instead, in the case shown in Listing 6, the smell was not satisfactorily resolved. ChatGPT removed the definition of the attributes `handler`, `mock`, and `runtime`. This way, the class needs to obtain their value using functions from the attributes' type classes. JDeodorant does not detect the smell anymore because it was triggered by the presence of those attributes, but the structure of the class is not really improved.

Listing 6. Portion of the God Class solved by ChatGPT without defining a new class.

```
// Smelly code
class KarateHttpMockHandlerTest {
      MockHandler handler;
      HttpServer server;
      FeatureBuilder mock;
      ScenarioRuntime runtime;

      void startMockServer() {
            // Sets attributes handler and server
      }
      FeatureBuilder background(String... lines) {
            // Sets and returns attribute mock
      }
      ScenarioRuntime run(String... lines) {
            // Sets and returns attribute runtime
      }
...}

// Cleaned code by ChatGPT
class KarateHttpMockHandlerTest {
      private HttpServer server;

      private FeatureBuilder background(String...
          lines) {
            // returns mock value directly
      }
      private HttpServer
          startMockServer(FeatureBuilder mock) {
            // returns the server value
      }
      private ScenarioRuntime
          runScenarioAndGetRuntime(String... lines) {
            // returns the runtime value directly
      }
...}
```

Two out of the five considered God Classes were not refactored by ChatGPT. God Classes contain many lines of code, which increases the length of the prompt. At the same time, refactoring a God Class requires the creation of a new class and the separation of responsibilities, which results in a lengthy response as well. This makes it much harder for the LLM, if it is not able to solve it in the first response, to retain all information provided and generated. This resulted in ChatGPT going in circles without ever solving the code smell.

The length of the God Classes considered in this study ranges between 81 and 515 lines of code (LOC). ChatGPT solved satisfactorily the two shortest ones (81 and 268 LOC) after the first prompt.

When dealing with a 320 LOC long class, ChatGPT was not able to provide correctly refactored code. After 12 iterations, ChatGPT generated some artifacts and potentially dangerous changes in the code (see Listing 6): public methods and attributes were made private. Although all test were passed successfully by the refactored class, the changes could cause problems.

The two classes ChatGPT did not refactor were 290 and 515 LOC long. The fact that longer code hinders the performance of LLM for code refactoring is in line with recent observations in the literature [13].

We tried to use GPT-4o, which allows for file embedding, but the required length for the generated output still hindered GPT's capabilities to generate correct code.

D. Refactoring Long Method

Similar to God Classes, Long Methods occur when a method `m` has multiple responsibilities; the smell can be solved by identifying the sub-section of the code that can be moved in a method `sm`, which is then called by `m` when needed. This solution was used by ChatGPT to correctly remove 5 of the 7 smells considered in this study.

ChatGPT tackled the last two methods by rewriting the code instead of creating a new method. An example is in Listing 7: this refactoring required multiple iterations and an indication of what triggered the smell detection. Although the code does not appear problematic, JDeodorant detected a Long Method smell just because there is a refactoring opportunity; in fact, the first section of the method, which is dedicated to building the `forwarded` string, can be extracted and moved in a new method. In the refactored code, JDeodorant did not detect the smell anymore because it was not able to find a refactoring opportunity. In the refactored code, three statements define three different variables, which are then used in `headers.set`. Since each string is built in one single statement, JDeodorant does not see an opportunity to extract a portion of the code.

While it is arguable if this method can be really considered "long," it is apparent that ChatGPT is able to refactor the code so that the—possibly false—alarm is no longer triggered.

Listing 7. Method flagged as Long Method by JDeodorant that ChatGPT solved by rewriting the code.

```
// Smelly code
String forwarded = /*forwarded*/
if(/*condition*/){
      forwarded = forwarded + ",";
} else {
      forwarded = "";
}
forwarded = forwarded + forwarded(uri, /*host*/);
headers.set("forwarded", forwarded);

// Code cleaned by ChatGPT
String existingForwarded = /*forwarded*/
String host = /*host*/
String newForwarded = forwarded(uri, host);

if(/*condition*/){
      headers.set("forwarded", existingForwarded +
          "," + newForwarded);
} else {
      headers.set("forwarded", newForwarded);
}
```

VI. Conclusions and Future Work

Table II reports the outcomes of the refactoring attempts by ChatGPT, for all the considered smell types. The table indicates in how many cases the smell was removed in a satisfactory way and how many prompts were necessary, how many unsatisfactory solutions were produced, and how many times no solution could be obtained.

TABLE II
Results obtained by ChatGPT for each code smell type.

Code Smell	# smells	Solved with # prompts 1	[2,5]	[6,12]	Bad solution	No solution
Type Checking	4	0	0	0	3	1
Feature Envy	14	6	1	0	5	2
God Class	5	2	0	0	1	2
Long Method	7	3	3	1	0	0

The fact that ChatGPT was able to generate correct solutions for all smells except Type Checking suggests that GPT 4o-mini is not able to propose polymorphism as a solution.

Concerning JDeodorant, we noticed that it yielded false positives, i.e., it identified smells in code that is actually of good (or acceptable) quality. This is particularly disturbing when JDeodorant is applied to the original code, because trusting JDeodorant can lead to instructing ChatGPT to address a code smell that is not actually present in the code. As a consequence, the LLM is compelled to generate new code that does not trigger the detection of the smell: this process can result not only in a degradation of code quality and structure, but also in subtle semantic changes that can be very dangerous if the available test cases are not sufficiently comprehensive to recognize them.

Alternatively, ChatGPT may fail to produce code that both passes all test cases and is not flagged by JDeodorant, even after several iterations. For instance, we asked ChatGPT to remove a God Class smell from a relatively short class (60 LOC), which had been incorrectly flagged as God Class by JDeodorant. After several iterations, ChatGPT was unable to generate a valid solution and began "going in circles" in its responses.

JDeodorant also yielded false negatives when used to check the refactored code: it did not recognize that some smells were still present in the refactored code. This implies that the process described in Figure 1 may not work well at the moment: practitioners may accept refactored code, deeming it of good quality, while the undetected smell is still there.

In conclusion, with the current tools, practitioners should carefully assess the issues flagged as code smells before employing LLMs for refactoring, and afterwards they should verify that the smells have actually been eliminated, rather than relying solely on the indications provided by the detection tool.

Next Steps

New LLMs are continuously released, each one more powerful than the preceding ones. Therefore, it is necessary to continuously repeat studies like the one reported here using the new LLMs. We experimented briefly with GPT-5: it appears to perform better than GPT 4o-mini, while suggesting better solutions than JDeodorant. For example, GPT-5 is able to apply polymorphism by defining the methods inside the enum, instead of defining a new class for each enum value. Therefore, repeating this study with GPT-5 (and possibly some other LLMs) is among our plans for future work.

Acknowledgment

The authors thank Samuele Ghirardello for cooperating in the experimental work.

References

[1] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
[2] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study," Automated Software Engineering, vol. 32, no. 1, 2025, p. 26.
[3] J. Cordeiro, S. Noei, and Y. Zou, "Llm-driven code refactoring: Opportunities and limitations," in 2025 IEEE/ACM Second IDE Workshop (IDE). IEEE, 2025, pp. 32–36.
[4] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells. conference paper," in 23rd IEEE International Conference on Software Maintenance and Evolution, ICSM, 2007.
[5] 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). IEEE, 2018, pp. 4–14.
[6] "Eclipse IDE," https://eclipseide.org/, accessed: 2025-09-05.
[7] "karate," https://github.com/karatelabs/karate, accessed: 2025-09-05.
[8] "Junit," https://github.com/junit-team/junit4, accessed: 2025-09-05.
[9] "Spring cloud gateway," https://github.com/spring-cloud/spring-cloud-gateway, accessed: 2025-09-05.
[10] "Gpt-4o mini: advancing cost-efficient intelligence," https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, July 2024, accessed: 2025-09-05.
[11] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Transactions on Software Engineering, vol. 35, no. 3, 2009, pp. 347–367.
[12] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in 2008 12th European conference on software maintenance and reengineering. IEEE, 2008, pp. 329–331.
[13] S. Dou, H. Jia, S. Wu, H. Zheng, M. Wu, Y. Tao, M. Zhang, M. Chai, J. Fan, Z. Xi, R. Zheng, Y. Wu, M. Wen, T. Gui, Q. Zhang, X. Qiu, and X. Huang, "What's wrong with your code generated by large language models? an extensive study," arXiv preprint arXiv:2407.06153, 2024.