

RefSearch: A Search Engine for Refactoring

Motoki Abe
School of Computing
Tokyo Institute of Technology
 Tokyo 152–8550, Japan
toki@se.c.titech.ac.jp

Shinpei Hayashi
School of Computing
Tokyo Institute of Technology
 Tokyo 152–8550, Japan
hayashi@c.titech.ac.jp

Abstract—Developers often refactor source code to improve its quality during software development. A challenge in refactoring is to determine if it can be applied or not. To help with this decision-making process, we aim to search for past refactoring cases that are similar to the current refactoring scenario. We have designed and implemented a system called RefSearch that enables users to search for refactoring cases through a user-friendly query language. The system collects refactoring instances using two refactoring detectors and provides a web interface for querying and browsing the cases. We used four refactoring scenarios as test cases to evaluate the expressiveness of the query language and the search performance of the system. RefSearch is available at <https://github.com/salab/refsearch>.

Index Terms—refactoring, search engine

I. INTRODUCTION

Refactoring is the process of restructuring the internal structure of source code without changing its external behavior [1]. Developers perform refactorings to improve the quality of their source code when adding new features to software or fixing bugs more efficiently [2]. One challenge in refactoring is to tackle the difficulty in determining whether or not to apply it. This also leads to the challenge of convincing team members, including managers and/or reviewers, when a refactoring is planned to apply [3]. Furthermore, developers are required to write high-quality and maintainable code. This has also sparked interest in educational methods for improving code quality through refactoring [4], [5].

Referring to real refactoring cases in past projects is an effective way to determine when and how to apply refactoring, and also to educate on its methods. Searching for refactoring cases that were conducted under similar conditions as the current project's source code can confirm whether such a refactoring was actually implemented, which can assist in the decision-making process. Moreover, one can learn how to refactor code through real refactoring example cases.

Collections of refactoring cases include a survey dataset by Silva *et al.* [2], where developers' refactoring reasons are attached to the cases, and the Refactoring Oracle [6] used in the evaluation of the refactoring detection tool Refactoring-Miner [7], [8]. When manually collecting refactoring cases from change histories, the precision is somewhat assured, but the human cost of building the dataset is high. Furthermore, the cost of verifying refactoring cases in projects not included in the existing dataset is high. Using refactoring detectors [8], [9] that can detect refactoring cases from change histories,

it is also possible to conduct searches based on their results. However, refactoring detectors are focused on detection itself and cannot be directly used to quickly identify cases that meet specific conditions from a large number of cases.

In this paper, we focus on the issues mentioned above and introduce a system named RefSearch. This system allows users to search for refactoring cases that meet specific conditions through a user-friendly interface. The main contributions of this paper can be summarized as follows:

- We have organized the information associated with each refactoring case to allow for a uniform search for refactoring cases obtained from multiple refactoring detectors.
- We have designed a query language that is both capable of searching for refactoring cases that meet specific conditions and easy to understand.
- We have designed and implemented a web search interface, making it easy to search for and view results.
- We have conducted a preliminary evaluation of the expressiveness of the query and the search performance of the system.

The remainder of this paper is structured as follows. In Section II, we clarify the motivations and issues related to searching for refactoring cases. In Section III, we explain the overall design of RefSearch and its components. In Section IV, we conduct a preliminary evaluation of RefSearch. Finally, in Section V, we conclude this paper and summarize future challenges.

II. BACKGROUND

A. Motivation

A difficulty in conducting refactoring is judging whether or not it should be applied because it does not involve adding new functionality or improving behavior [3]. When developers apply refactoring, they need to have appropriate justifications. We believe that referring to past similar refactoring cases can provide supporting evidence. Several use cases for conducting searches for refactoring cases are as follows:

- When renaming identifiers, developers may want to check if there have been similar renaming instances in the past to determine if the new name is really appropriate.
- When extracting common code from multiple methods, developers may want to examine past occurrences of duplicated code extraction to justify the extraction.

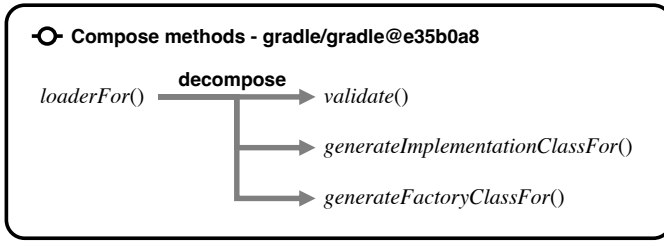


Fig. 1. Decomposing a method to multiple methods in Gradle.

- When extracting a meaningful code fragment from one method into a new method, developers may want to examine whether the size of the extraction is appropriate.
- When examining commit messages, developers may want to investigate what kinds of refactoring are included in commits labeled as refactoring.

For the details for the third case, suppose that a developer wants to determine the appropriate size for extracting code from a method consisting of 150 lines of code. In a previous commit titled “Compose methods” in the history of the Gradle project, the method *loaderFor* consisting of 168 lines was decomposed into three methods: *validate*, *generateImplementationClassFor*, and *generateFactoryClassFor*, as shown in Fig. 1¹. This refactoring might help the developer to convince other developers to conduct such an extraction refactoring.

We believe that searching for refactoring cases can be beneficial for various stakeholders in software development.

- *For practitioners:* As mentioned earlier, when refactoring planners are unsure whether a candidate refactoring is appropriate, finding real cases of similar refactorings in the same project or other projects can be helpful in making a decision and/or convincing team members to conduct the refactoring.
- *For beginners:* Although refactoring textbooks and refactoring catalogs provide explanations and application steps for refactorings, they lack rich examples. Specifically, examples in such books are often simplified and suitable for learning particular refactoring types, but are insufficient for understanding how such refactorings are performed in a real context. It would be beneficial for beginners to search for refactoring examples applied in the wild to learn practical applications of refactoring.
- *For researchers:* Real-world examples of refactoring can be a valuable resource for refactoring researchers. Researchers of refactoring tools and/or empirical studies on refactoring may need wild refactoring cases that meet specific conditions of their research context. Automated tools to search for such cases will facilitate their research activities.

B. Issues

When developers search for refactoring examples based on the aforementioned reasons, they can take one of the following

¹<https://github.com/gradle/gradle/commit/e35b0a8>

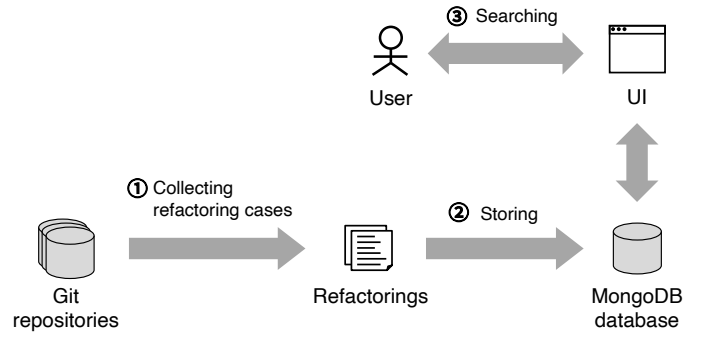


Fig. 2. Overview of RefSearch.

actions:

- 1) use a refactoring detector to detect examples and perform searches, or
- 2) use a search engine for code changes.

There are refactoring detectors available, such as RefDiff [10], [9], RefactoringMiner [7], [8], and Ref-Finder [11], that can detect past refactorings in projects. These detectors have high accuracy and can be used to find refactoring cases. However, refactoring detectors are designed for detection itself and cannot be directly used for quickly searching for refactoring cases that meet certain conditions from a large volume of source code change history.

Existing code change search methods are not suitable for searching refactoring cases. There are several code change searching methods available, such as GitHub Search and DiffSearch [12], [13]. However, GitHub Search does not specifically support searches for refactoring; instead, users must indirectly search for specific terms in commit messages. DiffSearch allows users to input custom queries that represent code fragments before and after changes, enabling them to search for chunks that represent the changes. While it is possible to specify detailed conditions for the code fragments before and after changes and conduct searches, it is unable to search for changes that span multiple chunks, making it unsuitable for locating refactoring cases.

III. REFSEARCH IN A NUTSHELL

A. Overview

To support the search and reference of refactorings, we have designed and implemented a refactoring example search system called RefSearch. The process of RefSearch is illustrated in Fig. 2. First, RefSearch collects refactoring cases using refactoring detectors. Next, it processes the collected cases into a searchable format and stores them in a MongoDB database. During this step, RefSearch builds indexes on important properties of the cases to enhance search performance. Finally, users can search for cases through a web interface. We will provide detailed explanations of each step.

B. Collecting Refactoring Cases

To collect refactoring cases from existing projects, we used two refactoring detectors: RefactoringMiner [8] and RefD-

TABLE I
INFORMATION OF A REFACTORING CASE

Property	Description	Example
<i>type</i>	Refactoring type	Extract Method
<i>description</i>	Description	Extracted method generateImplementation... from ...
<i>repository</i>	Git repository	https://github.com/gradle/gradle
<i>before</i>	Target code fragment before refactoring	
<i>.name</i>	Name	loaderFor (Class)
<i>.location.lines</i>	Number of lines	167
<i>.location.file</i>	File name	.../NamedObjectInstantiator.java
<i>after</i>	Target code fragment after refactoring	
<i>.name</i>	Name	generateImplementationClassFor (Class)
<i>.location.lines</i>	Number of lines	97
<i>.location.file</i>	File name	.../NamedObjectInstantiator.java
<i>commit</i>	Commit that contains the refactoring	
<i>.date</i>	Commit authoring date	2022-03-17T17.07.34Z
<i>.message</i>	Commit message	Polish 'NamedObjectInstantiator'
<i>.authorName</i>	Commit author	...
<i>.sha1</i>	Commit hash	e35b0a8c39182fdcbd11164eee028099657c0393
<i>.size</i>	Change size	
<i>.files.changed</i>	Number of changed files	2
<i>.lines.inserted</i>	Number of added lines	171
<i>.lines.deleted</i>	Number of removed lines	175
<i>.refactorings.total</i>	Number of refactorings in the commit	5
<i>extractMethod</i>	Details of Extract Method	
<i>.sourceMethodsCount</i>	Number of extracted methods	1
<i>.sourceMethodLines</i>	Lines in the method to be extracted	167
<i>.extractedLines</i>	Lines in the extracted method	97
<i>meta.tool</i>	Refactoring detector used	RefDiff

iff [9]. RefSearch automatically runs these two detectors and collects refactoring cases via the given Git repository URLs.

The output format of refactoring detectors varies depending on the specific detector used. To facilitate the search, RefSearch organizes and processes different outputs of refactoring detectors.

Table I presents the main properties of a refactoring case, along with an example refactoring case obtained from the Gradle project by applying the RefDiff detector. Each refactoring case is saved in a data structure of a hierarchical document that can be converted to the JSON format.

The information in each refactoring case includes the type of refactoring (*type*), a description of the operation (*description*), and information about the code fragments involved in the operation (*before* and *after*). In the case of the Extract Method refactoring detected by RefDiff, the code fragment before the refactoring (*before*) refers to the original method, while the code fragment after the refactoring (*after*) refers to the method extracted by applying this refactoring. For these code fragments, information such as name (*before.name*, *after.name*), lines of code (*before.location.lines*, *after.location.lines*), and file name (*before.location.file*, *after.location.file*) can be referenced. Note that the keys for information about code fragments may differ depending on the detector used, as the level of detail in the information about code fragments varies depending on the detector. Additionally, information about the commit in which the refactoring is found is also extracted. This commit information includes the commit date (*commit.date*), commit hash (*commit.sha1*), and size of the changes (*commit.size*). The size of the changes can be referenced as the number of files changed (*commit.size.files.changed*) and the number

```

query      = expr
characters = { ? visible characters ? }
word       = characters | '"' { characters } '"'
op         = '=' | '!=' | '~' | '<' | '<=' | '>' | '>='
expr       = logic [ '|' expr ]
logic      = primary [ "&" logic ]
primary    = word ' ' op ' ' word | '(' expr ')'

```

Fig. 3. Grammar of the query language (EBNF).

of inserted and deleted lines (*commit.size.lines.inserted*, *commit.size.lines.deleted*). Furthermore, the name of the detector used (*meta.tool*) can also be referenced.

C. Storing Refactoring Cases in a Database

To handle different document formats for refactoring cases, we utilized MongoDB. This database does not need a fixed schema to store and search for refactoring cases. To enhance the performance of common searches, we built indexes beforehand for the refactoring type (*type*) and commit date (*commit.date*).

D. User Interface

To search for refactoring cases with diverse data formats, we have designed a query language that is independent of specific data formats and easy to understand. The syntax of the designed query language is shown in Fig. 3. In the query, various conditions can be used to specify the properties of refactoring cases. These include exact match (using = and !=), partial match via regular expression (~), and numeric comparison (<, <=, >, and >=). Complex search conditions can be expressed using conjunction (&) and disjunction (|) operators. Several examples of queries are as follows:

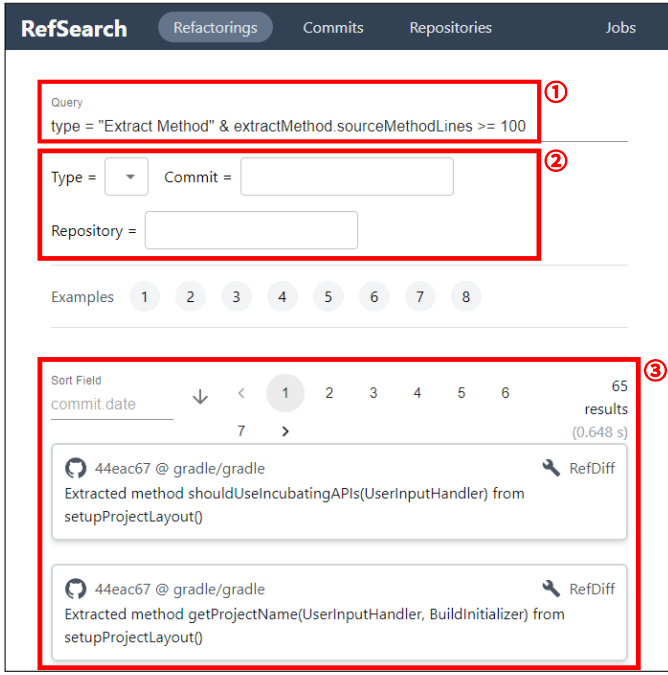


Fig. 4. Screenshot in searching for refactorings.

- `type = "Extract Method" & extractMethod.extractedLines >= 10` retrieves Extract Method refactorings where the extracted method contains ten or more lines of code.
- `type ~ /^Rename/ & rename.from ~ /^get/i & rename.to ~ /^retrieve/i` retrieves refactorings of a type starting with "Rename", i.e., rename refactorings, where the original name starts with "get" and the new name starts with "retrieve".

To make it easier for users to input queries, search for refactoring cases, and browse the search results, we have implemented a search interface that can be accessed through a web browser. A screenshot of the refactoring search page in RefSearch is shown in Fig. 4. Users can directly input a query ①. Additionally, RefSearch provides specific input fields ② for selecting the refactoring type, commit hash, and repository URL to facilitate the input of typical search items. After running the search, the search results are displayed at the bottom of the page ③. The results show an overview of the refactoring cases, including the repository name, the detector used, and a description of the refactoring.

Another view of the refactoring detail page is shown in Fig. 5. The top part of the page ④ displays basic information, including the repository name, commit hash, detector used, and description of the refactoring. The bottom part ⑤ displays the raw data of the refactoring case. Note that RefSearch does not provide a detailed code difference view that directly associates specific code changes with the refactoring operation. Users can still access the original commit link on GitHub to review the actual changes.

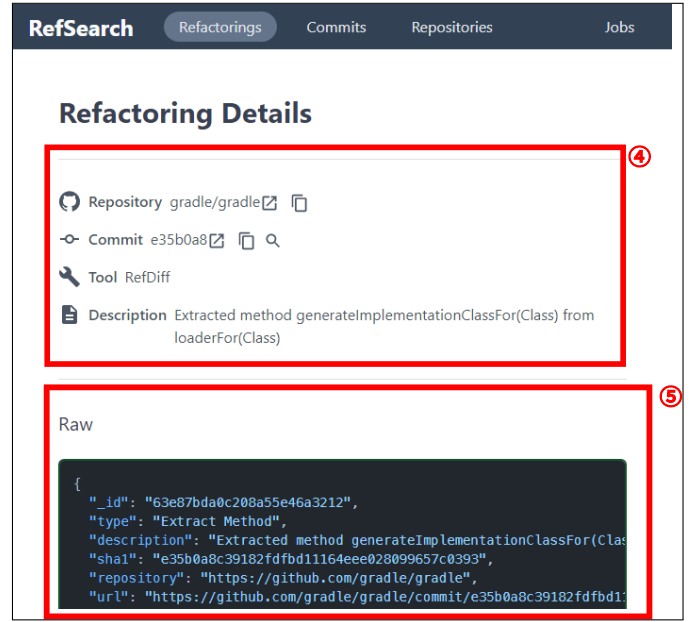


Fig. 5. Screenshot of the search result.

IV. PRELIMINARY EVALUATION

We want to verify whether developers can search for past refactorings related to the refactoring they are currently working on using RefSearch. As a first step, we conducted a preliminary evaluation to determine how easily RefSearch can search for refactorings that meet the specified conditions compared to existing search engines. We also consider the response speed as an important factor for a useful search engine. Therefore, we set the following two research questions (RQs):

- *RQ₁*: Can RefSearch find refactoring cases that meet the specified conditions better than GitHub Search and DiffSearch?
- *RQ₂*: What is the response speed of RefSearch?

A. *RQ₁: Search Efficiency*

1) *Study Design*: We assumed scenarios in which developers want to search for refactoring cases related to the refactoring that they are currently working on. We then evaluated how easily the search can be performed compared to two existing code change search engines: GitHub Search and DiffSearch [12], [13]. For our evaluation, we used Gradle² as the project. It has an adequate number of refactoring cases and sufficient history. We defined four conditions for searching refactoring cases. For each condition and search engine, we provided a search query that would help find refactoring cases matching the condition. We recorded the rank of the matching cases that met the condition in the search results. If no matching case was found within the top ten results, we considered it a failure. The assessment of whether a result matches with the condition was manually conducted by one of the authors.

²<https://github.com/gradle/gradle>

TABLE II
QUERIES USED

	RefSearch	GitHub Search	DiffSearch
#1	type ~ /^Rename/ & rename.from ~ /^get/i & rename.to ~ /^retrieve/i	(refactor OR rename) retrieve	<...>get();<...> → <...>retrieve();<...>
#2	type = "Extract Method" & extractMethod.sourceMethodsCount >= 2	(refactor OR extract) duplicate	<...> → <...>EXPR();<...>
#3	type = "Extract Method" & extractMethod.sourceMethodLines >= 100	(refactor OR extract) extract (large OR huge)	<...> → <...>EXPR();<...>
#4	type = "Extract Method" & commit.message ~ /extract/i	extract	<...> → <...>EXPR();<...>

TABLE III
RANK IN THE SEARCH RESULTS

	RefSearch	GitHub Search	DiffSearch
#1	1st / 2	N/A / 63	N/A / 0
#2	8th / 2,508	1st / 792	N/A / 432,151
#3	1st / 117	N/A / 1,143	N/A / 432,151
#4	8th / 443	1st / 1,143	N/A / 432,151

The prepared conditions for searching refactoring cases are as follows:

- #1 Renaming an identifier from “get...” to “retrieve...”.
- #2 Extracting a common part from multiple methods.
- #3 Extracting code from a method with more than 100 lines.
- #4 A self-affirmed Extract Method refactoring.

The search queries used for each condition in RefSearch are shown in Table II. For GitHub Search, we used search queries that involve words expected to appear in the commit messages because it is designed to search within commit messages. For DiffSearch, we used search queries to identify code fragments before and after the refactoring operations that are expected to be included in the changes because it expects to search changes of code fragments.

2) *Results*: Table III shows the rank and total number of results for each treatment. For Conditions 1 and 3, RefSearch produced a desired refactoring case as the top item. However, for Conditions 2 and 4, incorrect refactoring cases were included in the search results due to false positives in the output of refactoring detectors, resulting in a lower rank of 8th. In the case of GitHub Search, no matching cases were found for Conditions 1 and 3. Similarly, DiffSearch did not find any matching cases for any of the conditions.

RefSearch efficiently searched for refactoring cases that meet the given conditions compared to GitHub Search and DiffSearch.

B. RQ₂: Response Time

1) *Study Design*: We measured the response time for each query used in RQ₁. To answer this RQ, we assumed that developers would search only within their own projects and conducted searches within a single repository. We also evaluated searches across multiple repositories to account for the need to search for related refactorings from other projects. For searches within a single repository, we selected

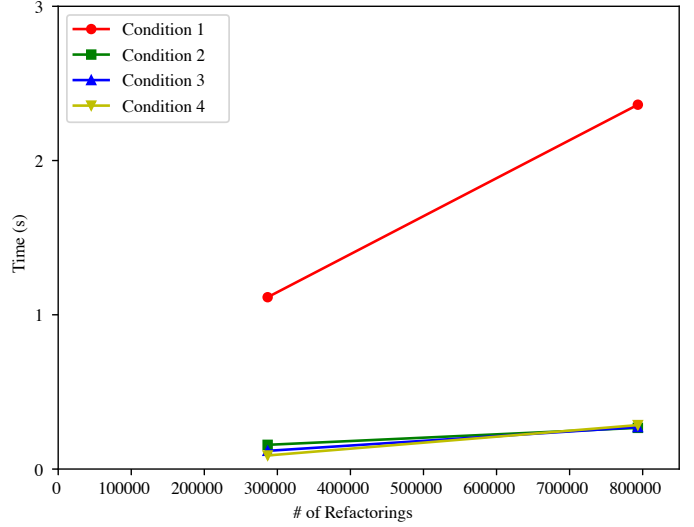


Fig. 6. Response time in search.

the Gradle project and targeted 286,686 refactoring cases detected from 101,704 commits. For searches across multiple repositories, we selected ten Java repositories with a sufficient number of commits. In total, we targeted 793,414 refactoring cases detected from a total of 285,783 commits in these ten repositories.

2) *Results*: The results of measuring the response time are shown in Fig. 6. For the query of Condition 1, which uses a regular expression, the response time was relatively slow, taking about 1 sec. for a single repository and about 2 sec. for multiple repositories. For the queries of Conditions 2, 3, and 4, the response time was below 0.5 sec. for both single and multiple repositories because the specified refactoring types allowed efficient utilization of pre-built indexes.

Even in repositories that contain an adequate number of refactoring cases, searches with approximately ten cases could be conducted within a realistic time frame of 1–3 sec. However, the response time increased as the number of stored cases to search increased.

C. Threats to Validity

Possible threats to validity can be listed up as follows.

- The queries used to answer RQ_1 were prepared by the authors. They may differ from the queries that ordinary developers can draft.
- In evaluating the four conditions in RQ_1 , there is a chance that the queries used in the existing methods were not optimally designed. A more suitable comparison of the methods could be achieved by using multiple queries for each condition and method, prepared by different individuals, and integrating the results of each query.
- In answering RQ_2 , we only measured the search speed in two scenarios: a single repository and ten repositories, and under four conditions. Therefore, the reliability of the conclusions concerning the trend of search speed might be insufficient.
- Gathering additional data points would enhance the reliability of the conclusions.

V. CONCLUSION

In this paper, we have designed and implemented a system called RefSearch. It enables users to search for refactoring cases that meet given specific conditions. Users can use a custom query language via a web interface to search for cases that satisfy the conditions. Our experiments have confirmed that RefSearch could effectively search for refactoring cases compared to two existing code change search engines: GitHub Search and DiffSearch.

Several future work can be listed up as follows.

- We plan to implement a detailed view of refactoring cases. Currently, RefSearch does not provide a detailed code difference representation like Refactoring-aware diff [14], which may not be sufficient for developers to understand the refactoring cases.
- As suggested in the discussion in RQ_2 , the search response time increased with the growth of data volume. Since simply indexing specific keys is insufficient for handling complex queries, redesigning the core part of the search using MongoDB may be necessary to improve the latency for complex queries involving large data volumes.
- The ease of understanding RefSearch queries has not been validated. By analyzing how developers actually formulate queries and evaluating their ease of use, we can identify potential issues of the query design.

ACKNOWLEDGMENTS

This paper is partly supported by JSPS Grants-in-Aid for Scientific Research JP22H03567, JP21H04877, JP21K18302, and JP21KK0179.

REFERENCES

- [1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [2] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? Confessions of GitHub contributors,” in *Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 2016, pp. 858–870.
- [3] T. Sharma, G. Suryanarayana, and G. Samarthayam, “Challenges to and solutions for refactoring adoption: An industrial perspective,” *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.
- [4] H. Keuning, B. Heeren, and J. Jeuring, “A tutoring system to learn code refactoring,” in *Proc. 52nd ACM Technical Symposium on Computer Science Education (SIGCSE 2021)*, 2021, pp. 562–568.
- [5] —, “Student refactoring behaviour in a programming tutor,” in *Proc. 20th Koli Calling International Conference on Computing Education Research*, 2020, pp. 1–10.
- [6] “Refactoring oracle,” <http://refactoring.enss.concordia.ca/oracle/>, (Accessed on 01/27/2023).
- [7] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinianian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *Proc. 40th IEEE/ACM International Conference on Software Engineering (ICSE 2018)*, 2018, pp. 483–494.
- [8] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.
- [9] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, “RefDiff 2.0: A multi-language refactoring detection tool,” *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2021.
- [10] D. Silva and M. T. Valente, “RefDiff: Detecting refactorings in version histories,” in *Proc. 14th IEEE/ACM International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 269–279.
- [11] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, “Template-based reconstruction of complex refactorings,” in *Proc. 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010, pp. 1–10.
- [12] L. D. Grazia, P. Bredl, and M. Pradel, “DiffSearch: A scalable and precise search engine for code changes,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2366–2380, 2023.
- [13] L. Di Grazia, “Efficiently and precisely searching for code changes with DiffSearch,” in *Companion Proc. 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*, 2022, pp. 313–315.
- [14] R. Brito and M. T. Valente, “RAID: Tool support for refactoring-aware code reviews,” in *Proc. 29th IEEE/ACM International Conference on Program Comprehension (ICPC 2021)*, 2021, pp. 265–275.