CIRef: A Tool for Visualizing the Historical Data of Software Refactorings in Java Projects

Marcos Gênesis, Maykon Nunes
marcosgenesis@alu.ufc.br
maykon.nunes@alu.ufc.br
Federal University of Ceará
Quixadá, Brazil

Carla Bezerra
carlailane@ufc.br
Federal University of Ceará
Quixadá, Brazil

Anderson Uchôa
andersonuchoa@ufc.br
Federal University of Ceará
Itapajé, Brazil

Mairieli Wessel
mairieli.wessel@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

Context: Refactorings are actions developers do not often do in a standard way. One of the reasons is the lack of visualizations in current tools capable of identifying refactorings. Code visualizations can help developers make decisions about analyzing code quality and possible code refactorings. **Objective:** We present CIRef, a tool for visualizing the historical data of refactorings in Java projects. For a particular project, CIRef provides a wide range of visualizations including customizable rankings of the importance of different refactoring types, duels between developers to understand their profiles, and a timeline of the number of refactorings performed. **Method:** To validate the acceptance and perceived usefulness of CIRef, we conducted a study with eight developers using the Technology Acceptance Model (TAM). **Results:** The results indicate that 75% of the participants agreed with using the tool in the future and found it easy to use. **Conclusions:** Beyond supporting developers in visualizing historical refactoring data, CIRef also has the potential for educational purposes. It can help teachers visualize the history of refactorings performed by students, especially in educational environments focused on programming and maintaining Java projects.

Video link: https://figshare.com/s/99c9e2ca3fb227b649a1

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **Human-centered computing → Visualization techniques**.

KEYWORDS

Software visualization, Refactoring, Java projects

ACM Reference Format:
Marcos Gênesis, Maykon Nunes, Carla Bezerra, Anderson Uchôa, and Mairieli Wessel. 2023. CIRef: A Tool for Visualizing the Historical Data of Software Refactorings in Java Projects. In *XXXVII Brazilian Symposium on Software Engineering (SBES 2023), September 25–29, 2023, Campo Grande, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3613372.3613419

SBES 2023, September 25–29, 2023, Campo Grande, Brazil

ACM ISBN 979-8-4007-0787-2/23/09…$15.00
https://doi.org/10.1145/3613372.3613419

1 INTRODUCTION

Refactoring is a practice that involves modifying and restructuring existing code without changing its functionality [12]. The primary goal of refactoring is to improve non-functional aspects of the code, such as reducing complexity and increasing modularity, reusability, and maintainability [16]. These quality aspects counterbalance the high cost and development effort required by the refactoring process (e.g., in a medium-sized project– with around 30,000 lines of code–the refactoring time is approximately six weeks [15]). Refactoring helps developers to complete maintenance tasks (i.e., fix bugs, implement new features) and eliminate code smells [18].

Refactorings are rather infrequent or performed in a non-standard way, mostly because current refactoring tools are not straightforward and might frustrate developers who attempt to use them [13]. Furthermore, existing tools often lack diverse visualizations of refactorings, which are crucial for promoting increased adoption of refactoring practices in projects [6]. AlOmar et al. [5] identified that developers who are responsible for the majority of refactorings in a project tend to document less of their refactoring activities compared to developers who perform refactorings sporadically. This suggests that the currently existing tools do not support the developers who perform the most refactorings.

In this work, we developed and evaluated CIRef, a tool specifically designed for visualizing the historical data of refactorings in Java projects. In the tool, it is possible to visualize the refactorings of one or more projects represented by absolute numbers and by developers, in addition to the number of refactorings according to time and the prioritization of the types of individualized refactorings for each project. Therefore, practitioners and researchers can use our tool to understand the history and types of refactorings carried out over time in projects.

2 BACKGROUND

RefactoringMiner [20] is a tool that identifies refactorings performed over time, given the complete history of a project. Despite being built by the same authors as the previous tool, RefactoringMiner contains a list of different types of refactorings it can find in projects written in Java. Some of the refactorings identified by tool include: (1) *Extract Method*, (2) *Inline Method*, (3) *Rename Method*, (4) *Move Method*, (5) Move Attribute, (6) *Merge Variable*, (7) *Move and Rename Method*, and (8) *Reorder Parameter*. The list above represents 8 of the 90 different types of refactorings that this tool supports identifying, making the tool most suitable for our work. Another advantage of this tool is that it is not limited to being just a library that can be added to IDEs, but also works as a native API in Java

projects. Finally, the tool returns its results in the same format, facilitating integration with other services.

3 RELATED WORK

Bogart et al. [7] extended the JDeodorant tool with fine-grained visualization of multiple refactoring operations suggested to the user by the primary refactoring tool, enabling developers to better understand the impact of applying these refactorings. Our tool differs from this work by using the history of refactorings implemented in the project to demonstrate different types of visualizations.

Brito and Valente [8] proposed the RAID tool, which partially automates the code review process in *Pull Requests* within the GitHub tool. This tool can identify refactoring operations in these *Pull Requests* and alleviate the cognitive effort required in the code review action. The tool uses another tool, RefDiff [17], to find refactorings. In this work, we will identify the refactorings already implemented in the project and show them in a specific platform instead of showing the data in the Github tool itself.

Tsantalis et al. [20] proposed the RefactoringMiner tool, which captures all refactorings implemented between a child commit and a parent commit per the GitHub repository control flow. The created tool supports about 90 types of refactorings and is widely a reference in the context of tools for detecting refactorings. In this work, we will use this tool as a provider of data on refactorings in the analyzed projects. However, due to the support for many types of refactorings, we will also categorize these supported types to facilitate the visualization of more compact types of graphs.

4 CIREF AT A GLANCE

CIRef[1] is a web tool that aims to provide views about the historical data of refactorings in Java projects. It offers a user-friendly interface that helps developers and project managers to know the profile of refactorings performed by project collaborators, the number of refactorings over time, and compare refactorings performed by these developers. The license used by CIRef is the MIT Open Source [2]. The key features of the tool are: (F1) allows login with the GitHub account; (F2) allows the user to select multiple repositories for extracting the refactorings; (F3) shows data from one repository at a time; (F4) user can select the time window in which the refactorings occurred; (F5) user can adjust the priority of the types of refactorings in the platform; (F6) user can select two developers to compare the refactorings performed by them; (F7) synchronize the data whenever a project is selected; (F8) user can view the most refactored files in a project; and (F9) user can view all types and subtypes of refactorings performed on a project.

4.1 CIRef Architecture

Figure 1 illustrates the three main architectural components of the CIRef tool. The diagram is organized from the highest level, which is the front-end, where the end user will have direct contact with the tool, to the lowest level, where the GitHub API and the refactoring extraction API are present.

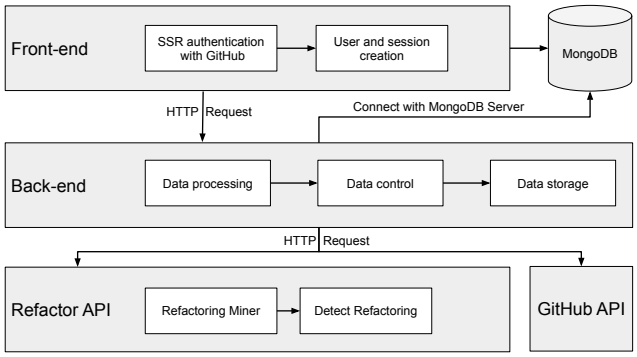

Figure 1: CIRef architecture

Refactor API. At the lowest level layer, we create an API, named Refactor API[2]. This API serves as a wrapper for using RefactoringMiner to extract the refactorings from the projects. The name wrapper is given by the decomposition strategy chosen for this layer. This API is available through a microservice connected to the back-end layer of the tool. By adopting this type of separation between the back-end layers and the wrapper of the RefactoringMiner tool, this API can be maintained, and new functionalities implemented and scaled to meet high request demands, regardless of the layer it is connected to. This decomposition strategy was chosen because this service is the main layer of the tool being proposed in this work. Therefore, failures in this service have to be minimal, ensuring that the service remains stable and available. The API was developed using Kotlin programming language, the Spring Boot framework, and Maven, the package manager for Java applications [1]. We choose these technologies due to their ease of deployment and the need to interact with RefactoringMiner, which runs in the Java Virtual Machine.

Back-end. The application's back-end[3] is responsible for working with both the data from the Refactor API and the data from the GitHub API. In addition to managing all the routes that will serve the data directly to the front-end, this application layer will handle the management of the database, avoiding duplication of records of the refactorings found and inserting them into the database. CIRef's back-end was developed with the NestJS [3], which works internally with strategies that guarantee more security and scalability for the application. This is due to its module structure, where each module is built independently of other modules and can even be reused in other NestJS applications.

Front-end. The front-end[4] of this tool is developed using the ReactJS library, which is very popular among the web application development community. Allied with this library, the CIRef tool was developed using the NextJS framework that offers us essential functionalities regarding sensitive information, such as a GitHub code repository. For this reason, in the tool proposed in this work, all application authentication is handled with a concept called Server Side Rendering (SSR). This concept boils down to executing some

[1] https://ciref.vercel.app/

[2] https://github.com/leanresearchlab/ciref-refactor
[3] https://github.com/leanresearchlab/ciref-api
[4] https://github.com/leanresearchlab/ciref

business rules, even before the tree of elements of the WEB application is assembled. Allowing validations to be performed, calls to external resources such as APIs. In this application, we use this feature to handle the authentication of users on the platform, keeping all information safe.

4.2 CIRef Features

Figure 2 represents an overview of all the features of the CIRef tool. We describe all features as follows.

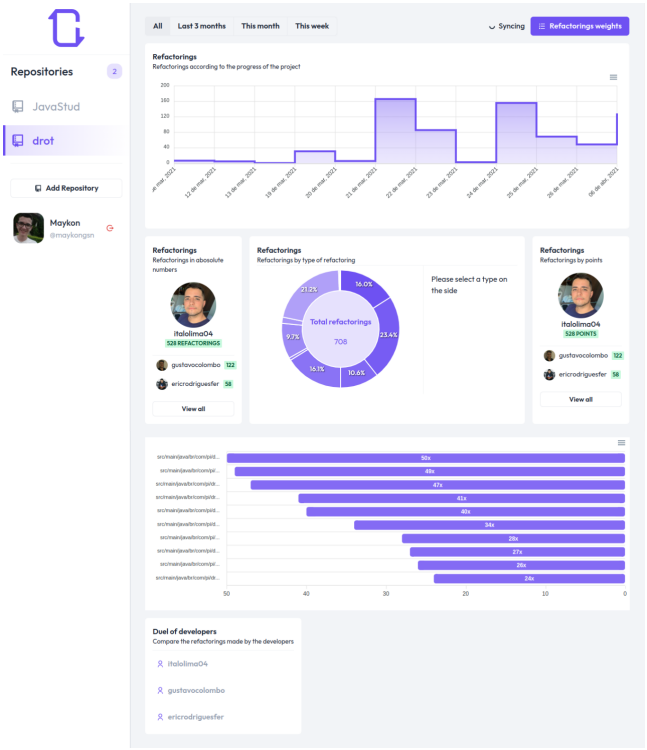

Figure 2: CIRef overview

Loading project data from GitHub. Initially, the user must access the tool using the GitHub login. In the first use, the data comes directly from the Github API and is filtered to show only projects whose main language is Java, a supported language for extracting the refactorings performed in the tool proposed in this work. The user can perform two main actions: (1) select one or more repositories to which the tool has access; (2) click on the Submit button so that these preferences are saved in the database.

View the number of refactorings over time. This feature refers to the visualization of all refactorings performed within a period, which is the entire duration of the project by default. In Figure 3, it is possible to observe the presence of a graph of lines, with the lines adapted to show the correct steps of the progression of the numbers. This means that the line chart in this feature will not show sharp angles, as is the default for this type of chart.

This feature is interesting for identifying unusual patterns in the flow of refactorings performed on projects. In Figure 3, you can recognize particular spikes in the number of refactorings around

Figure 3: Number of refactorings over time

specific dates. If those dates happened to be end-of-sprint dates, we have information that developers are doing more refactorings near the end of the cycle. This information could serve as a basis for some decision-making within the project, and with the help of other features described later, even the developers who most performed these refactorings could be identified.

View of the absolute number of refactorings per contributor. The main information of this feature is the ranking of developers who performed the most refactorings in a project according to a time window. In this visualization, no type of criterion is considered for counting the numbers of refactorings, only their absolute numbers. That way, you can find out which developer performed the most refactorings on the project. However, this information needs to be interpreted carefully, because this information does not differentiate between simple and complex refactorings, it only brings the absolute number of refactorings.

View of the most recurrent types of refactoring in the project. With this feature, it is possible to visualize the percentage of all types of refactoring performed within a project. Figure 4 shows how the tool implements this view. We chose to make this representation through a donut-type graph. It was necessary to group the refactorings into subtypes so that this donut view would not be damaged. However, the original data on the types of refactorings were not lost. When hovering the mouse over the partitions of the donut chart, it is possible to visualize the subtypes of the refactoring present in each category represented in the chart.

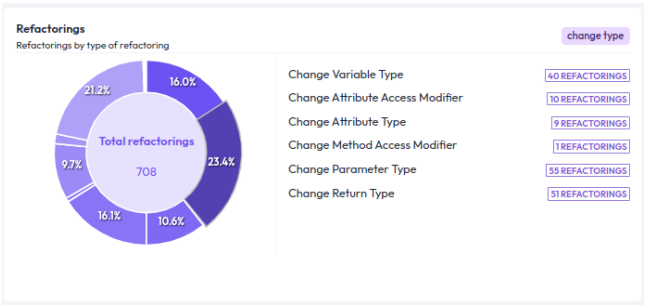

Figure 4: Number of refactorings by type

View of developers' score based on prioritization of refactorings. In this feature, the developer assigns different weights to each type of refactoring, adjusted to the context of the repository. This allows a developer to prioritize certain refactoring types over others based on their importance to the project. In the tool's default view, all refactorings are weighted equally. However, this view

can be enhanced by prioritizing the types of refactoring activated by clicking the button. When clicked, the button opens a drawer, showing all categories of refactorings (Figure 5). Initially, with the equivalent weight of 1x - this means that 1 refactoring is equal to 1 point. At this time, the user can freely prioritize the refactorings according to what is most important for the project.

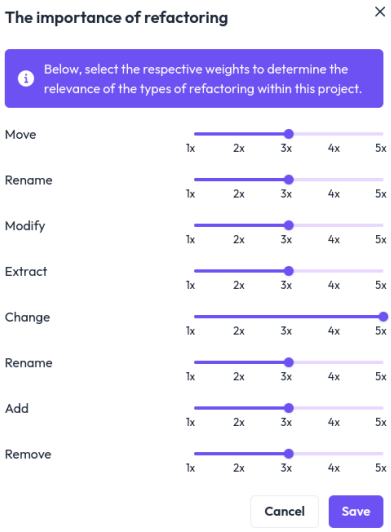

Figure 5: Drawer to perform prioritization of refactorings

View of the most refactored files. This feature provides a view of the 10 most refactored files within a specific time window of a project (Figure 2). It allows users to identify the relative path of the most modified files and the types of refactorings that are predominantly applied to those files. This information enables users to infer insights about the data visualization, identifying if developers face challenges or difficulties implementing certain functionalities within specific files.

Duel between developers. Figure 6 represents the feature that compares the refactorings of the project developers using the idea of challenge [4]. This comparison makes it possible to identify the refactoring profile that developers most perform. This feature also serves as a complement to the feature related to different weights to refactorings, where it is possible to visualize the distribution of the types of refactoring performed by a given developer, which added to the weight assigned to the types of refactoring can help to find the best profiles of developers who apply refactorings to projects.

5 STUDY DESIGN

Using the Goal-Question-Metric (GQM) template [9] our goal can be further defined as: **analyze** the proposed tool; **for the purpose of** characterization; **with respect to** the acceptance and perceived usefulness; **from the viewpoint of** developers; **in the context of** Java software projects. Our research question is: *What are the acceptance and perceived usefulness from the point of view of developers?* Gathering developer feedback is crucial to validate the proposed tool before sharing it with the community. Therefore, we

Figure 6: Developer duel over refactorings

conducted a survey designed to obtain initial results on developers' acceptance and perception of the usefulness of the proposed tool.

5.1 Survey Design

To achieve our goal, we have designed a survey based on the Technology Acceptance Model (TAM) [11]. This model is suitable to capture the user's acceptance of a given technology [21]. Additionally, this instrument is frequently used in software engineering literature, e.g., [10, 19]. The TAM involved gathering feedback to assess three acceptance model constructs: usefulness, ease of use, and intention to use. Perceived usefulness is defined as the extent to which a technology is expected to improve a potential user's performance. Perceived ease of use is defined as the amount of effort required to effectively use technology. Intention to use is measured by a person's intention to use the technology.

Table 1 shows our survey which the questions are organized to measure each of the three main constructs of TAM: perceived usefulness (UP); ease of use (FUP), and self-predicted future use (AUF). We used a 5-point Likert scale [14] to measure participants' agreement with each statement, ranging from *Strongly disagree* to *Strongly agree* and including a neutral value.

Table 1: Scale items for measuring UP, FUP and AUF

Perceived Usefulness (UP)
UP1. Using the proposed tool would help me to have more control over my code.
UP2. Using the proposed tool would reduce time spent on less important refactorings.
UP3. Using the proposed tool would improve the quality of my work/projects.
UP4. Using the proposed tool would increase my productivity.
UP5. Using the proposed tool would give me more knowledge about refactorings.
UP6. Using the proposed tool would enable me to gain time at work/projects,
UP7. The visualizations provided by the proposed tool are useful in my day-to-day work.
UP8. Checking the visualizations provided by the proposed tool would allow me to better compare my performance with other developers.
UP9. The visualization provided by the platform on the number of refactorings performed over time is a differential in relation to other existing tools
UP10. I would find the proposed tool useful.
Perceived Ease of Use (FUP)
FUP1. My interaction with the proposed tool is easy to understand.
FUP2. The proposed tool shows important information about refactoring.
FUP3. By using the tool I get information to implement other types of refactoring.
FUP4. Overall, I find the proposed tool useful for my work.
Self-prediction of Future Use (AUF)
AUF1. Assuming the proposed tool would be available, I predict that I will use it in my work/projects
AUF2. I find the tool's insight into refactoring more interesting than other tools/plugins.

Target survey population. Our target population consists of developers who meet two criteria: (1) a minimum of 2 years of

experience in software development for industrial projects, and (2) experience working with the Java programming language on projects. These criteria were selected to ensure that the participants have practical experience in real-world projects and familiarity with the demand for refactoring tasks in industrial settings.

We recruited 8 developers for our study. Among them, 50% had more than 4 years of experience in software development and Java projects, 37.5% had 3 years of experience, and the remaining 12.5% met the minimum requirement of two years of experience.

Tool Usage and Survey Execution. Prior to conducting the survey, we provided all participants with an introduction to the proposed tool. Specifically, we demonstrated all the functionalities described in Section 4.2. To illustrate these functionalities, we used a real software project hosted on GitHub as input for the tool. Subsequently, we organized a raffle to determine the order in which the eight participants would use the tool. Each participant then had an individual session to explore the tool. We emphasized that participants were only given instructions for the initial configuration of the tool and connecting it to their repositories. Once set up, each participant was free to utilize the tool in any way they wished, and we encouraged them to explore all the features it offered. There was no specific time limit for this exploration phase. After participants finished using the tool, we requested them to complete a survey that included the questions presented in Table 1.

6 EVALUATION RESULTS

We present the results of the survey about the CIRef tool's perceived usefulness, ease of use, and prediction of future use as follows.

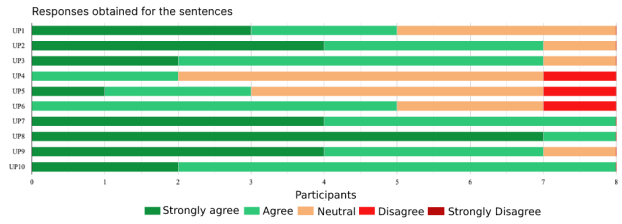

Figure 7: Responses to perceived usefulness items

Perceived Usefulness. Most participants found the CIRef useful. We present each item's detailed results in Figure 7. In particular, all participants agreed or strongly agreed on the usefulness of the visualizations provided by the tool in their daily work (UP7), the possibility of comparing developers' performance in terms of refactoring (UP8), and the overall usefulness of CIRef (UP10). Such observations demonstrate that the planned functionalities for the tool were carefully designed and are feasible. Furthermore, it was emphasized that the visualizations present in the tool are different compared to other tools that address the theme of refactorings.

We also observed a neutral stance among participants regarding items UP4 and UP5. This suggests that although the tool provides visualizations of the historical context of refactorings, 62.5% of participants believe that these visualizations will not have a significant impact on their productivity in both work environments and personal projects. On the other hand, 25% of the participants believe utilizing these visualizations will increase productivity. From these observations, we can infer that the proposed tool may serve primarily as a data visualization tool rather than a tool that directly enhances developers' productivity. This highlights an opportunity for future work to focus on enhancing the tool's functionality to achieve more positive outcomes.

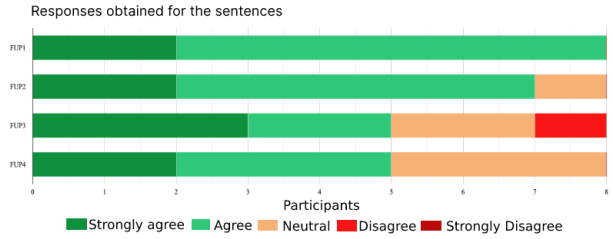

Figure 8: Responses to perceived ease of use items

Perceived Ease of Use. In Figure 8 shows the answers' distribution per item related to the ease of use. More than 62.5% of the participants agreed or strongly agreed with the items. All participants agreed the interaction with the tools is easy to understand (FUP1). This indicates the effectiveness of the refactoring visualizations combined with the historical context of the project provided by the tool. Only one participant disagreed that the tool provides information for implementing other types of refactorings (FUP3).

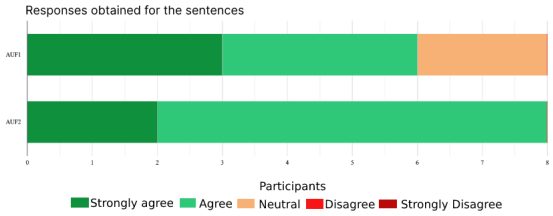

Figure 9: Responses to self-prediction future use items

Self-prediction of Future Use. Figure 9 reports self-predicted future use of the tool. We observed that 6 (75%) participants agreed or strongly agreed that they would use CIRef in the future (AUF1). All participants agreed that the visualizations provided by the tool are more appealing than the existing visualizations on other platforms (AUF2). These results indicate that developers are interested in using the proposed tool in their projects.

7 TOOL IMPACTS

CIRef has several benefits for monitoring refactorings, education on refactoring practice, and research in mining software repositories.

Regarding *education about refactoring practices*, CIRef can be used as a *monitoring tool to track the refactorings employed* by students. Teachers can guide students in conducting code review activities to enhance the system's design through refactoring applications. By utilizing CIRef as a monitoring tool, teachers can assess the progress and effectiveness of students' refactoring efforts. Furthermore, teachers can prioritize specific types of refactorings

for instruction, encouraging their application in the system and promoting a deeper understanding of those particular refactorings.

CIRef can also be used for software developers and managers to monitor the intensity of refactoring in their projects. This information can be used to encourage cycles of code quality improvement and identify developers who actively practice continuous improvement through refactoring. Finally, regarding mining software repositories research, the CIRef offers researchers *insights into the application of refactoring in software projects* through visualizations of key aspects. These visualizations include information on the most refactored files, trends about employed refactorings over time, and initial characterizations of developer profiles based on their employed refactorings. By providing synthesized information, CIref eliminates the need for extensive preprocessing of raw data obtained from refactoring detection tools.

8 LIMITATIONS

During the validation process with participants, some limitations were identified regarding the construction of the tool and its potential future use. Approximately 37.5% of them faced difficulties in understanding the status of the application while it was synchronizing the refactoring data of the selected projects. This highlights the need for careful planning on how the platform handles situations when the data is either empty or outdated. Addressing this limitation can contribute to improving the user experience and ensuring the tool's reliability in real-world scenarios.

Another limitation of the tool is its reliance on the RefactoringMiner tool as the sole source of data for searching the history of refactorings in Java projects. The refactorings provided by the tool are not filtered to select only those that preserve the code's behavior. Additionally, there is a lack of specific validation for each visualization used in the tool. Without dedicated validations, it becomes difficult to confirm the effectiveness of the chosen graph types in conveying the intended information. This may impact users' understanding of the refactorings and limit the tool's usability. Finally, another limitation is the limited sample size of only eight participants in the tool validation stage. A small sample size reduces the generalizability of the findings and limits the ability to draw definitive conclusions about the tool's effectiveness and usability. A larger, more diverse participant pool would provide more reliable and representative results.

9 CONCLUDING REMARKS

In this study, we introduce CIRef, a tool designed for visualizing the historical data of refactorings in Java projects. It provides developers with a timeline visualization of the number of refactorings performed allowing the identification of recurring periods where the number of employed refactorings increases or decreases, highlighting potential patterns and trends. The tool also allows a simultaneous performance comparison between two developers (i.e., a duel between developers), and prioritization of the types of refactoring by assigning different weights.

We have evaluated the CIRef tool, by surveying 8 developers. We based the survey on the TAM model to assess the perceived usefulness, ease of use, and self-prediction of future use. The evaluation indicated a strong demand for the tool among developers. The key strengths identified were the tool's visualizations and ease of use, which received high acceptance from the evaluation participants. However, opportunities for improvement were also identified, including the lack of incentives for practicing refactorings based on the on-screen information, as well as disagreement among developers regarding the time-saving benefits of adopting the tool. We also planning to validate the tool by targeting teachers, managers, and researchers in future studies.

ACKNOWLEDGMENTS

This research was partially funded by PIBITI-UFC and FUNCAP (BP5-00197-00042.01.00/22).

REFERENCES

[1] [n. d.]. Maven. https://maven.apache.org/. (Accessed on 01/19/2023).
[2] [n. d.]. Mit. https://opensource.org/licenses/mit-license.php. (Accessed on 01/19/2023).
[3] [n. d.]. Nestjs. https://nestjs.com/. (Accessed on 01/19/2023).
[4] Cláuvin Almeida, Marcos Kalinowski, Anderson Uchôa, and Bruno Feijó. 2023. Negative effects of gamification in education software: Systematic mapping and practitioner perceptions. *Inf. Softw. Technol.* 156 (2023), 107142.
[5] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. 2021. Behind the scenes: On the relationship between developer experience and refactoring. *J. Softw.: Evol. Process* (2021), e2395.
[6] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. Increasing the Trust In Refactoring Through Visualization. In *42nd International Conference on Software Engineering Workshops (ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 334–341. https://doi.org/10.1145/3387940.3392190
[7] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. Increasing the Trust In Refactoring Through Visualization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. New York, NY, USA, 334–341.
[8] Rodrigo Brito and Marco Tulio Valente. 2021. RAID: Tool Support for Refactoring-Aware Code Reviews. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 265–275.
[9] Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. 1994. Goal question metric paradigm. *Encyclopedia of software engineering* 1, 528-532 (1994), 6.
[10] Hui Hui Chen, Ming Che Lee, Yun Lin Wu, Jing Yao Qiu, Cheng He Lin, Hong Yong Tang, and Ching Hui Chen. 2012. An analysis of moodle in engineering education: The TAM perspective. In *International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, H1C–1.
[11] Fred D Davis. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly* (1989), 319–340.
[12] Martin Fowler. 2018. *Refactoring*. Addison-Wesley Professional.
[13] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.* 40, 7 (2014), 633–649.
[14] R. Likert. 1932. *A Technique for the Measurement of Attitudes*. Number Nº 136-165 in A Technique for the Measurement of Attitudes. Archives of Psychology.
[15] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 535–546.
[16] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139.
[17] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Trans. Softw. Eng.* 47, 12 (2021), 2786–2802.
[18] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870.
[19] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. 2016. Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering*. 273–284.
[20] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Softw. Eng.* 48, 3 (2022), 930–950.
[21] Mark Turner, Barbara Kitchenham, Pearl Brereton, Stuart Charters, and David Budgen. 2010. Does the technology acceptance model predict actual use? A systematic literature review. *Inf. Softw. Technol.* 52, 5 (2010), 463–479.