A New Visualization for Refactorings in Pull Requests: Proposal and Preliminary Evaluation

Alicia Paiva
State University of Ceara - UECE
Fortaleza, Brazil
alicia.paiva@aluno.uece.br

Matheus Paixao
State University of Ceara - UECE
Fortaleza, Brazil
matheus.paixao@uece.br

Adriely Silva
State University of Ceara - UECE
Fortaleza, Brazil
adri.silva@aluno.uece.br

Victor Timbó
State University of Ceara - UECE
Fortaleza, Brazil
victor.timbo@aluno.uece.br

Alysson Diniz dos Santos
Federal University of Ceara - UFC
Fortaleza, Brazil
alysson@virtual.ufc.br

Matheus Chagas
State University of Ceara - UECE
Fortaleza, Brazil
matheus.chagas@aluno.uece.br

Abstract

Code review is a fundamental activity in collaborative software development and is commonly performed through Pull Requests (PRs) on GitHub. Refactoring consists of rewriting code without changing its external behavior, improving its internal quality. In PRs, refactorings are typically presented in traditional diff-based interfaces as added and removed lines, making their intent difficult to understand. Consequently, reviewers must mentally reconstruct the performed transformations, increasing cognitive effort and potentially impacting review effectiveness. To address this gap, we envisioned RavnCR (**R**efactoring **A**ware **V**isualizatio**N** for **C**ode **R**eview), an approach that aims to improve the understanding of refactorings during PR reviews through refactoring identification and specialized visualizations. RavnCR aims to extend the traditional GitHub review interface by explicitly highlighting refactoring-related changes while preserving the familiar review workflow. To validate RavnCR's proposed visualization, we performed a preliminary quali-quantitative study involving developers with experience in code review. The evaluation assessed the approach's usability, effectiveness, and efficiency. Participants reported that the visualization made the refactorings more explicit, helped them identify and interpret the structural code changes, and made the review process faster and more intuitive. However, some visual elements, especially color-based highlights, require further refinement to reduce ambiguity and improve developer experience.

Keywords

Developer Experience, Code Review, Refactoring

1 Introduction

Software development is a complex and collaborative activity that involves multiple stages, ranging from implementing new features to the continuous evolution of code [22]. In modern software projects, this collaboration is supported by development platforms such as GitHub, which enable version tracking, contribution integration, and communication among developers [6]. In GitHub, proposed code changes are submitted through Pull Requests (PRs).

A PR involves two main roles. The author, who submits the changes, and the reviewers, who evaluate them. Reviewers provide comments, questions, or suggestions, and the author revises the code accordingly. This cycle may repeat until the changes are ready to be integrated into the codebase, requiring reviewers to interpret and understand the code written by other team members.

Developer productivity depends not only on technical skills, but also on the tools used during software development [18]. Development tools mediate developers' interaction with software artifacts. In the context of GitHub, the PR visualization interface is based on a textual diff model, in which differences between code versions are presented line by line. The removed lines are highlighted in red, while the added lines are highlighted in green. [9].

Refactoring consists of rewriting code without changing its external behavior, improving its internal quality [8]. Most refactorings do not occur in isolation, but with other code changes [17], making their understanding during reviews more difficult. This scenario is challenging in GitHub, where the review interface presents code changes as removed and added lines. In the case of refactorings, the reviewer needs to reconstruct the meaning of the change, increasing cognitive load and negatively impacting productivity [18].

To address this gap, we envisioned RavnCR (**R**efactoring **A**ware **V**isualizatio**N** for **C**ode **R**eview), an approach that combines refactoring identification with visualizations that make refactoring explicit during code review. RavnCR will integrate refactoring detection mechanisms with an enhanced review interface that highlights and contextualizes refactoring-related changes. The goal of this paper is to investigate and validate RavnCR's proposed visualization of refactorings in code review. Hence, RavnCR is currently represented as a high-fidelity interactive prototype developed using Figma. This prototyping approach enables the evaluation of usability, effectiveness, and efficiency while preserving a realistic PR review experience. The current prototype extends the traditional GitHub interface focusing on two common refactoring operations: *Extract Method* and *Pull Up Method*.

We conducted a preliminary quali-quantitative study with five developers experienced in code review, using SUS [5], think-aloud, and open-ended questions to gather quantitative and qualitative evidence. The preliminary evaluation of RavnCR suggests that explicitly visualizing refactorings can support developers during code review. Results show positive perceptions as participants were able to identify refactoring types, understand the relationship between the original and modified code, and explain the performed changes. Nevertheless, the findings also indicate that some visual elements, such as color highlights, should be improved through captions, contextual instructions, or clearer visual cues.

The main contributions of this work are:

- A high-fidelity interactive prototype that improves the visualization of two refactoring operations: *Extract Method* and *Pull Up Method*.
- A preliminary quali-quantitative evaluation to assess the usability, effectiveness, and efficiency of the proposed approach.
- A replication package containing the prototype, study materials, collected data to support transparency. [16]

2 Background

2.1 Code Review and Refactoring

Collaborative development platforms are essential in modern software development, enabling teams to work in a coordinated manner. GitHub stands out as the widely used platform [6]. Proposed code modifications are generally submitted through Pull Requests (PRs). A PR is a collaborative artifact that centralizes discussions, comments, and technical decisions, while also presenting the comparison between two versions of the code. The PR interface in GitHub displays differences between versions using a textual diff model, in which removed lines are highlighted in red and added lines in green. This visualization enables the identification of code changes. However, it does not explicitly convey their semantic meaning.

Code review is a quality assurance mechanism capable of detecting potential defects before the integration of code changes [1]. Developers analyze modifications proposed by other team members, evaluating aspects such as clarity, correctness, adherence to design patterns, and possible architectural impacts. In developers' routines, collaborative activities such as code review are an essential part of software development, being performed frequently and requiring cognitive effort, since they demand the understanding of the proposed changes and their impact on the system [13].

Refactoring is the process of modifying the internal structure of a software system while preserving its external behavior, to improve its internal quality [8]. In practice, refactoring can occur at different times. In many cases, it is performed when developers need to modify an existing functionality while taking the opportunity to improve the code [20]. Thus, most refactorings do not occur in isolation, but instead in conjunction with other code changes [17], making their understanding during code reviews more difficult.

Reviewers must not only identify what lines were added or removed, but also understand whether the proposed changes represent behavioral modifications, structural improvements, or both. When refactorings are mixed with feature implementation or bug fixes, this task becomes more challenging, since the textual diff does not clearly differentiate changes that preserve behavior from those that change system functionality. Hence, reviewers may need to spend more effort interpreting the intention behind the modifications and assessing their impact on the codebase. Therefore, explicitly supporting the visualization and comprehension of refactorings in PRs can contribute to a more effective review process.

2.2 Developer Experience

Developer eXperience (*DX*) refers to the perceptions, emotions, and interactions that developers have with the tools, processes, and environments involved in software development activities [18]. Unlike traditional approaches that primarily assess development performance through quantitative metrics, DX emphasizes the human aspects of software engineering, including cognitive, emotional, and social factors that influence how developers perform their work [7].

A positive Developer eXperience has been associated with improved productivity, efficiency, and satisfaction, as it reduces friction in development workflows and enables developers to focus on higher-value tasks [18]. In this context, tools characterized by poor visual clarity, complex interaction flows, or excessive cognitive demands may increase the effort required to understand software artifacts and perform development tasks [19].

2.3 Prototyping

Prototypes are simplified representations of software systems used to explore design alternatives, validate requirements, assess usability, and collect early user feedback before full implementation [10, 14, 23]. They are especially relevant for new visualization and interaction techniques, as they allow researchers to evaluate not only technical feasibility but also whether users can understand the presented information and perform their tasks more efficiently and with less effort.

This work adopts a prototype-based strategy to validate RanvCR's proposed approach to visualize refactorings in PRs. Rather than developing a production-ready tool, the goal is to validate the feasibility of the proposed interface and investigate whether explicitly presenting refactorings can improve reviewers' understanding of code changes.

2.4 Usability

Usability is a quality criterion of Human-Computer Interaction (HCI). It is related to the ease with which users can learn, use, and understand an interface, as well as to their level of satisfaction when interacting with a system. In general, usability aims to assess whether a solution enables users to perform their tasks in a clear, simple, and appropriate way, considering aspects such as learnability, efficiency of use, error prevention, and satisfaction [3].

According to ISO 9241-11, usability is defined as the degree to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [11]. From this perspective, effectiveness is related to the correct completion of the proposed tasks, efficiency refers to the resources required to achieve these goals, and satisfaction concerns the user's subjective perception during system use.

3 Proposal for a New Refactoring Visualization Approach

RanvCR's interface was designed for integration in GitHub's PR review context, allowing reviewers to explicitly see refactoring alongside the modified code. The new visualization includes a set of visual and interaction features to support comprehension. One of these features is the use of color to highlight refactoring-related information. Purple was adopted as the main color because it differs from the conventional colors commonly used in code diffs, such as green and red. This distinction emphasizes that refactorings represent structural changes, rather than additions or removals.

Another feature of the proposed visualization is the presence of navigation mechanisms. During a PR review, refactoring may involve multiple code fragments, methods, classes, or files. Without explicit support, reviewers may need to manually search for related changes across the diff, increasing cognitive effort and making the review more difficult. To reduce this effort, the approach provides navigation elements that allow reviewers to move between detected refactorings and inspect each operation individually.

The approach also includes an on/off interaction structure. This feature allows reviewers to enable or disable the refactoring visualization according to their needs. When enabled, the interface highlights and organizes information related to refactorings. When disabled, the interface returns to the conventional PR visualization. This was included to preserve user control and avoid overloading the interface with additional information when not necessary.

As previously mentioned, refactoring modifies code while preserving its externally observable behavior, improving code quality and maintainability [8]. Among the commonly used refactorings in software maintenance and evolution are *Extract Method*, *Inline Method*, *Pull Up Method* and *Move Method* [15]. This work focuses on two of them: *Extract Method* and *Pull Up Method*.

The *Extract Method* refactoring moves a cohesive code fragment from an existing method to a new method with a meaningful name, improving readability and reducing method complexity. When shown only as a traditional diff, this refactoring may appear merely as removed and added lines, requiring the reviewer to mentally connect the extracted code block with the new method. In our visualization, the involved code fragments are highlighted in purple, making the extracted block and its corresponding method explicit. Visual emphasis is given to the added lines that define the new method, allowing reviewers to quickly locate the destination of the extracted code. Rather than indicating that these lines were added, the purple color has a semantic meaning to identify code elements that participate in the same refactoring operation and visually communicates the relationship between the original fragment and the newly created method. Figure 1 displays the proposed *Extract Method* visualization. The code used in this example is discussed in Section 4.

The *Pull Up Method* refactoring moves a method from subclasses to a common superclass, reducing duplication and strengthening the class hierarchy. In a traditional diff, this change appears only as removals and additions, requiring reviewers to infer that the method was moved. Our visualization highlights the refactored fragments in purple and provides an interactive class diagram that enables navigation between classes and their corresponding code, making the method movement easier to understand. Figure 2 presents this visualization, using the code discussed in Section 4.

4 Empirical Evaluation

This section describes the empirical evaluation conducted to assess the proposed approach for visualizing refactorings in PR. The goal of this evaluation is to investigate how the proposed visualization supports developers during code review, especially when they need to identify, understand, and interpret refactoring-related changes. This empirical study evaluates the usability, effectiveness, and efficiency, as discussed in Section 2.4. For this purpose, we employed the high-fidelity prototypes presented in Figures 1 and 2. To guide the evaluation, we defined the following research questions:

- *RQ1: What is the perceived usability of RavnCR according to the SUS score?*
- *RQ2: What was the participants' perceived effectiveness using RavnCR's visualization?*
- *RQ3: What was the participants' perceived efficiency using RavnCR's visualization?*

4.1 Empirical Methodology

4.1.1 Examples. To evaluate the proposed visualizations, we crafted examples of *Extract Method* and *Pull Up Method* refactorings. For *Extract Method*, we considered the method `calculateTotal`, belonging to the `Manager` class. In the initial version of the code, presented in Listing 1, the `calculateTotal` method is responsible for calculating the total value of a set of items and applying a discount.

```
public double calculateTotal() {
    double total = 0;
    for (Item item : items) {
        total += item.getPrice() * item.
            getQuantity();
    }
    if (total > 100) {
        total *= 0.9;
    }
    return total;
}
```

Listing 1: Initial version of the calculateTotal method.

To improve code organization, the calculation of the total value was extracted into the `calculateItems` method, while the discount logic was extratced to the `applyDiscount` method. The original method became responsible for coordinating these operations, as presented in Listing 2. The way this refactoring appears in the visualization can be observed in Figure 1.

```
public double calculateTotal() {
    double total = calculateItems();
    return applyDiscount(total);
}
private double calculateItems() {
    double total = 0;
    for (Item item : items) {
        total += item.getPrice() * item.
            getQuantity();
    }
    return total;
}
private double applyDiscount(double total) {
    if (total > 100) {
        total *= 0.9;
    }
    return total;
}
```

Listing 2: Refactored version after applying *Extract Method*.

In the *Pull Up Method* refactoring scenario, the example involves a superclass, `Employee`, and two subclasses, `Developer` and

Manager. Initially, both subclasses independently implement the calculateBonus method, which is responsible for calculating the employee's bonus based on salary, as shown in Listing 3.

```
// Developer.java
public double calculateBonus() {
    return salary * 0.20;
}
// Manager.java
public double calculateBonus() {
    return salary * 0.20;
}
```

Listing 3: Initial version of the calculateBonus method.

The duplication represents a maintainability problem. Therefore, the *Pull Up Method* refactoring was applied, moving the duplicated method to the Employee superclass, as illustrated in Listing 4. The refactoring is illustrated in Figure 2.

```
// Employee.java
public double calculateBonus() {
    return salary * 0.20;
}
```

Listing 4: Refactored version after applying *Pull Up Method*.

4.1.2 Data Collection. Data collection was conducted through a quali-quantitative procedure combining structured questionnaires, task execution, and the think-aloud technique. It enabled the analysis of task completion, participants' interpretations of the interface, encountered difficulties, and visualization elements that supported or hindered understanding.

Participants were recruited through convenience sampling based on availability and compliance with the eligibility criteria. Initially, participants answered a characterization questionnaire was used to collect information about their previous experience with software development, code review, GitHub, and refactoring. This information was important to interpret the results, since familiarity with code review and refactoring concepts could influence participants' perceived usability.

After the characterization step, participants interacted with the prototypes using the code examples in Listing 1, 2, 3 and 4. During this interaction, they were asked to identify and understand the visualized refactorings while verbalizing their thoughts, doubts, expectations, and interpretations. The think-aloud technique allowed us to examine their reasoning and identify usability issues that might not emerge from closed-ended questions alone.

The interaction sessions were conducted in person and lasted approximately 35 minutes each. At the end of the session, participants answered an evaluation questionnaire composed of closed and open questions. The closed questions characterize the System Usability Scale (SUS) [5], which provided a standardized usability score, while the open questions captured participants' positive impressions, difficulties, suggestions, and overall perceptions of the proposed visualizations

4.1.3 Data Analysis. The System Usability Scale (SUS) [5] was used to evaluate the perceived usability of RavnCR. The responses were converted into contributions from 0 to 4, summed, and multiplied by 2.5, resulting in a score from 0 to 100. Scores above 70 are considered acceptable, between 50 and 70 marginal, and below 50 unacceptable [2]. Therefore, the result indicates high perceived usability and contributes to answering RQ1.

The qualitative data consisted of the participants' responses to the open-ended questions and their verbalization during the think-aloud sessions. The researchers manually transcribed the audio recordings produced during the sessions. Although manual transcription required more effort, it allowed the researchers to preserve the context of each statement, identify expressions directly related to the participants' interactions with the prototype, and maintain consistency between the recorded sessions and the resulting textual data. In total, the transcription process resulted in approximately 13 pages and 4375 words. The transcriptions were subsequently combined with the participants' written responses to the open-ended questions, forming the qualitative corpus analyzed in this study.

The analysis was conducted through repeated and systematic readings of the complete qualitative corpus. During this process, excerpts related to participants' perceptions, difficulties, behaviors, and reactions while interacting with the prototype were identified and extracted. Statements concerning the participants' ability to understand the presented refactorings were associated with effectiveness and used to answer RQ1, and excerpts describing the effort, time, or difficulties involved in completing the activities were associated with efficiency and used to answer RQ2.

4.2 Results

4.2.1 Participant Profile. The usability evaluation involved six participants, including one pilot participant whose data were excluded from the analysis. Thus, the final sample comprised five Computing students or professionals with three to five years of programming experience. Python was used by all participants, followed by JavaScript/TypeScript (80%), Java (60%), C/C++ (40%), and Go (20%). Regarding Object-Oriented Programming, 80% reported intermediate knowledge and 20% advanced knowledge.

Participants had limited experience with refactoring: 20% reported no knowledge, 60% basic knowledge, and 20% intermediate knowledge; 60% rarely performed refactorings and 40% did so occasionally, while none used specific visualization tools. For both *Git* and *GitHub*, 60% reported basic knowledge, 20% intermediate knowledge, and 20% advanced knowledge. Participation in *Pull Requests* was frequent for 40%, occasional for 40%, and rare for 20%. Regarding code reviews, 40% had never performed them, while 20% reported rare, occasional, and frequent participation, respectively.

4.2.2 RQ1: What is the perceived usability of RavnCR according to the SUS score? As shown in Figure 3, most participants perceived the tool as having acceptable usability. P1, P2, P3, and P5 obtained SUS scores of 87.5, 90.0, 92.5, and 95.0, respectively, indicating excellent usability, whereas P4 obtained 65.0, indicating marginal usability. P4 also reported greater confusion and need for assistance, suggesting that some interface elements or interaction flows require improvement. Despite this lower score, the results were predominantly positive, with an average SUS score of 86.0.

4.2.3 RQ2: What was the participants' perceived effectiveness using RavnCR's visualization? In the open-ended questions, P1 stated: 'I found the way the refactoring visualizations were presented on

Figure 1: RavnCR's Extract Method Visualization

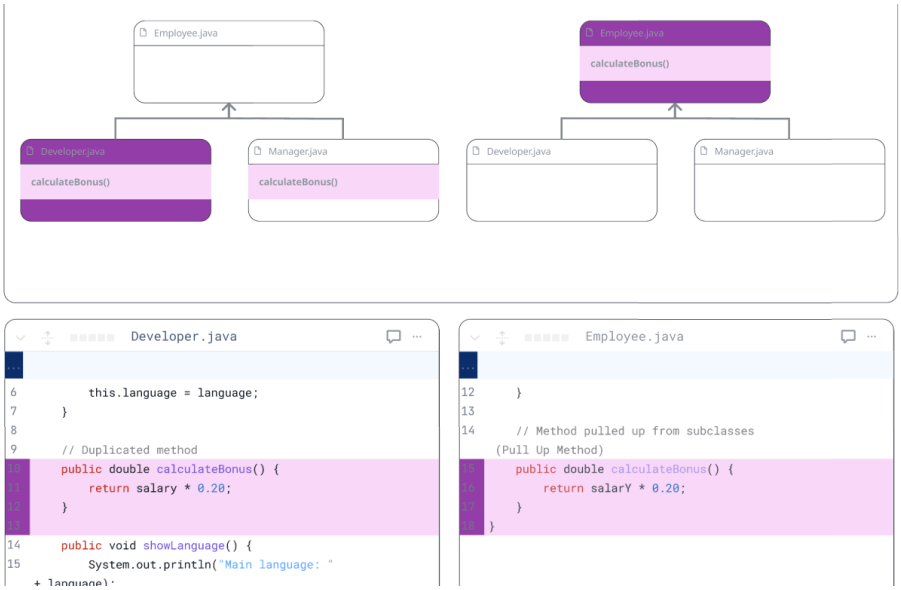

Figure 2: RavnCR's Pull Up Method Visualization

the screen intuitive. Even without much knowledge of refactorings, it was clear to understand which type of refactoring was being shown." P3 added: 'In particular, the visualization of Extract Method helped me clearly perceive what was and what was not part of the refactoring."

During the think-aloud session, participants also demonstrated their understanding of the changes. P1 explained: 'Basically, it took what was calculate total, which was a single function, and divided it into several functions to calculate total and call other functions that do the same thing, but in a more organized way." P4 observed: 'Apparently everything was inside one method, and here it was transformed into several methods: apply discount, total, getitem count." Similarly, P5 stated: "The total calculation was very basic before. In the new version, several steps and specific elements were added, such as the discount itself and auxiliary functions to support the calculation."

However, P4 reported ambiguity regarding the highlighting: "I did not understand the purple color. I do not understand why it is purple. The other changes were clearer, but here there are two highlighted regions."

Overall, the open-ended and think-aloud data show that the visualization supported refactoring understanding, although some elements require clearer explanations.

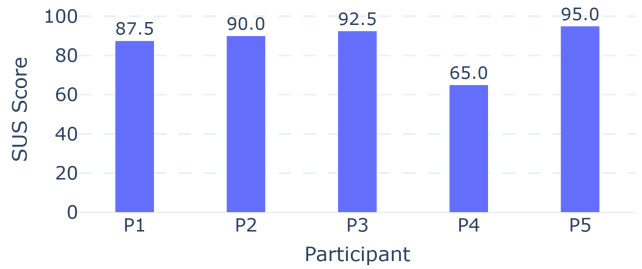

Figure 3: Individual SUS scores by participant.

4.2.4 RQ3: What was the participants' perceived efficiency using RavnCR's visualization? The open-ended answers state the perception of efficiency. P2 said "I think that because it is more intuitive, it would even make the process faster and with good quality." Similarly, P3 highlighted "It makes the experience more visual and speeds up the understanding of the code." , while P5 discussed "I believe it would make code reviews faster and more intuitive." These reports indicate that participants associated the visualization not only with clarity, but also with a reduction in the cognitive effort required to identify and interpret refactorings.

The think-aloud data also provided evidence of this relationship with efficiency. P1 was able to quickly understand the transformation performed, stating that the original function had been divided into smaller and more organized functions, and concluded that "it was possible to understand". P2 explored the interface and progressively identified the elements presented, such as modified files, comments, before-and-after code, and navigation between code sections, which suggests that the interface guided the interpretation of the changes. P5 said "So I can switch between them to see directly through the diagram what happened, the changes."

This result suggests the visualization has the potential to make code review more efficient. Thus, it indicates that the proposal may contribute to a faster review process with lower interpretive effort.

5 Threats to Validity

The number of participants was limited, which restricts the generalization of the results to populations of reviewers. In addition, the study focused only on two refactoring types, namely *Extract Method* and *Pull Up Method*. To mitigate this, the study collected both quanti-qualitative data, combining closed-ended questions, open-ended questions, and think-aloud observations.

The evaluation was conducted using high-fidelity prototypes designed in Figma rather than a fully implemented tool integrated into GitHub. The use of high-fidelity prototypes helped approximate the interaction that users would have with an integrated review interface. Although the prototype was not a fully implemented GitHub extension, it reproduced the main visual elements and interaction flow proposed by the approach.

Another threat to validity concerns the assessment of participant satisfaction. Since the evaluation involved a single, short interaction with a prototype in a controlled setting, it was not possible to measure satisfaction resulting from prolonged or repeated use of the proposed approach. The collected data therefore reflect participants' immediate perceptions after the evaluation rather than their satisfaction in a real code review workflow over time.

6 Related Work

`RAID` (*Refactoring-Aware and Intelligent Diff*) is a code review support tool designed to make refactorings explicit during the review process [4]. The approach automatically detects refactoring operations in PRs and reorganizes the traditional GitHub diff through a browser extension that highlights these transformations. RAID's approach is solely centered on the textual reorganization of diffs, keeping the same color scheme, which provides limited support for the visual and semantic interpretation of structural transformations.

`REFACTORINSIGHT` is an IntelliJ plug-in that automatically identifies refactorings and separates them from behavioral changes [12]. However, the proposal is restricted to the IDE environment and does not investigate how alternative visualization strategies could further facilitate the comprehension of refactoring operations during code review.

`CIRef` is a web-based tool that supports the exploration of refactoring histories in Java projects [21]. The approach detects refactorings in GitHub repositories and presents them through timeline-based visualizations and file-ranking mechanisms. Nevertheless, CIRef focuses on retrospective analysis of software evolution rather than supporting the code review process itself.

7 Conclusion

Code review is an essential practice in modern software development, acting as a quality gateway for integrating changes into software projects. However, when PRS include refactorings, traditional diff-based visualizations may make the review process more difficult, since reviewers need to mentally reconstruct the semantic meaning of the changed code. This work addressed this problem by proposing, an approach for making refactorings explicit during code review through dedicated visualizations.

To investigate the perceived usability of the proposed approach, we evaluated high-fidelity prototypes for *Extract Method* and *Pull Up Method* through closed-ended questions, open-ended questions, and think-aloud observations. Overall, participants considered the visualization useful, easy to understand, and more effective than a traditional diff for identifying and interpreting refactorings, suggesting that it can reduce comprehension effort and improve the code review experience. However, some participants faced difficulties with specific visualization elements and required additional guidance.

As future work, we intend to implement RavnCR as an integrated tool in GitHub, and evaluate it in realistic development scenarios. We also plan to extend the approach to support additional refactoring types and conduct studies with a larger and more diverse group of developers. In addition, future evaluations should investigate participant satisfaction through longer-term and repeated use of the tool in real code review workflows.

Artifact Availability

All artifacts, including the prototype link, study results, and research forms, are available in https://zenodo.org/records/21879661

References

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 712–721.

[2] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction* 24, 6 (2008), 574–594.

[3] Simone Diniz Junqueira Barbosa, Bruno Santana da Silva, Milene Selbach Silveira, Isabela Gasparini, Ticianne Darin, and Gabriel Diniz Junqueira Barbosa. 2021. *Interação Humano-Computador e Experiência do Usuário*. Autopublicação.

[4] 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. doi:10.1109/ICPC52881.2021.00033

[5] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[6] Panuchart Bunyakiati and Usa Sammapun. 2019. Open-Source Projects and their Collaborative Development Workflows. *arXiv preprint arXiv:1909.00642* (2019).

[7] Fabian Fagerholm and Jürgen Münch. 2012. Developer experience: Concept and definition. In *2012 international conference on software and system process (ICSSP)*. IEEE, 73–77.

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1 ed.). Addison-Wesley Professional.

[9] GitHub. 2026. *Reviewing proposed changes in a pull request.* https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/reviewing-proposed-changes-in-a-pull-request

[10] Stephanie Houde and Charles Hill. 1997. Chapter 16 - What do Prototypes Prototype? In *Handbook of Human-Computer Interaction (Second Edition)* (second edition ed.), Marting G. Helander, Thomas K. Landauer, and Prasad V. Prabhu (Eds.). North-Holland, Amsterdam, 367–381. doi:10.1016/B978-044481862-1.50082-0

[11] International Organization for Standardization. 2018. *ISO 9241-11:2018 Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts.* International Organization for Standardization, Geneva, Switzerland.

[12] Zarina Kurbatova, Vladimir Kovalenko, Ioana Savu, Bob Brockbernd, Dan Andreescu, Matei Anton, Roman Venediktov, Elena Tikhomirova, and Timofey Bryksin. 2021. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[13] André N Meyer, Earl T Barr, Christian Bird, and Thomas Zimmermann. 2019. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering* 47, 5 (2019), 863–880.

[14] Jakob Nielsen. 1993. *Usability Engineering*. Academic Press.

[15] Daniel Oliveira, Wesley K.G. Assunção, Alessandro Garcia, Ana Carla Bibiano, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. 2023. The Untold Story of Code Refactoring Customizations in Practice. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM.

[16] Alicia Paiva. 2026. Replication Package. https://doi.org/10.5281/zenodo.21879661.

[17] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *Proceedings of the 25th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 176–185. doi:10.1109/ICPC.2017.38

[18] Abdul Razzaq, Jim F. Buckley, Qin Lai, Yun Ting, and Goetz Botterweck. 2024. A Systematic Literature Review on the Influence of Enhanced Developer Experience on Developers' Productivity: Factors, Practices, and Recommendations. *Comput. Surveys* (2024). doi:10.1145/3687299

[19] Beatriz Santana, Lidivânio Monteiro, Bianca Santana de Araújo Silva, Glauco Carneiro, Sávio Freire, José Amancio Macedo Santos, and Manoel Mendonça. 2024. Psychological Safety in Software Workplaces: A Systematic Literature Review. (2024).

[20] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. 858–870.

[21] Marcos Gênesis da Silva. 2022. CIREF: uma ferramenta para visualização do contexto histórico de fatorações em projeto java. (2022).

[22] Ian Sommerville. 2011. *Engenharia de Software* (9 ed.). Pearson Prentice Hall, São Paulo.

[23] Miriam Walker, Leila Takayama, and James A Landay. 2002. High-fidelity or low-fidelity, paper or computer? Choosing attributes when testing web prototypes. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 46. Sage Publications Sage CA: Los Angeles, CA, 661–665.