# Spike – A code editor plugin highlighting fine-grained changes

Ronald Escobar
*Exact Sciences and Engineering Research Center (CICEI)*
*Universidad Católica Boliviana "San Pablo"*
Cochabamba, Bolivia
ronaldescobarj@gmail.com

Juan Pablo Sandoval Alcocer
*Department of Computer Science, School of Engineering*
*Pontificia Universidad Católica de Chile*
Santiago, Chile
juanpablo.sandoval@ing.puc.cl

Hagen Tarner
*University of Duisburg-Essen*
Essen, Germany
hagen.tarner@paluno.uni-due.de

Fabian Beck
*University of Bamberg*
Bamberg, Germany
fabian.beck@uni-bamberg.de

Alexandre Bergel
*RelationalAI*
Switzerland
https://bergel.eu

***Abstract*—Information about source code changes is important for many software development activities. As such, modern IDEs, including, *IntelliJ IDEA* and *Visual Studio Code*, show visual clues within the code editor that highlight lines that have been changed since the last synchronization with the code repository. However, the granularity of the change information is limited to a line level, showing mainly a small colored icon on the left side of the lines that have been added, deleted, or modified.**

**This paper introduces Spike, a source code highlighting plugin that uses the font color to visually encode fine-grained version difference information within the code editor. In contrast to previously mentioned tools, Spike can highlight insertions, deletions, updates, and refactorings all in a same line. Our plugin also enriches the source code with small icons that allow retrieving detailed information about a given code change. We perform an exploratory user study with five professional software engineers. Our results show that our approach is able to assist practitioners with complex comprehension tasks about software history within the code editor.**

***Index Terms*—Code highlighting; software evolution; software visualization**

Artifact: https://doi.org/10.5281/zenodo.7026727

## I. Introduction

Syntax highlighting is a common feature of source code editors. It typically applies predefined font colors to the code to provide visual cues about syntax, for instance, keywords, control structures, constants, and literals. The underlining goal of traditional syntax highlighting is to help distinguish various language elements and to ease program comprehension. However, a recent study has shown that there is no evidence that standard syntax highlighting improves developers' abilities to comprehend source code [1].

We argue that an alternative application of source code highlighting is helping developers in analyzing code changes. Analyzing source code changes is an important activity during software development. It is known that developers frequently refer to source code changes to better understand the code and make choices while programming [2]. A study based on 217 developer interviews reveals that at least 61% of them review the code history a few times a day, and 85% consider that code history is important in their development activities [2]. Among the questions that developers frequently ask themselves when programming are *When, and how this code was changed or inserted?*, *What recent changes were made?*, and *What else changed when this code was inserted or changed?* [3].

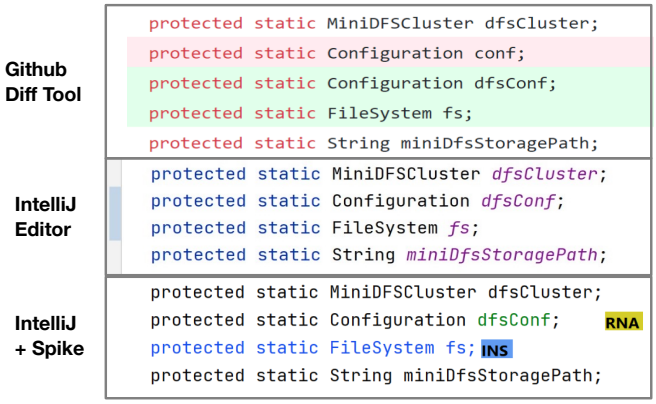


Fig. 1. Source code change example in: GitHub, IntelliJ, and Spike

Traditional source code difference approaches typically display both versions of the modified lines, duplicating the information. For instance, Figure 1 gives an example of a code modification in the unified GitHub code difference tool. This modification involves a renaming of a variable and insertion of a line. These changes are shown by GitHub as one line deletion (red lines) and two line additions (green lines). *IntelliJ IDEA* (similar as other editors like *Visual Studio Code*) flag the modified lines by adding a color on the left on the side-bar, with line-based details similar to the GitHub representation available on demand. However, it is difficult to identify fine-grained changes within a line, for instance, a simple variable renaming.

This paper presents Spike, a change-based source code highlighting mechanism to show fine-grained version differ-

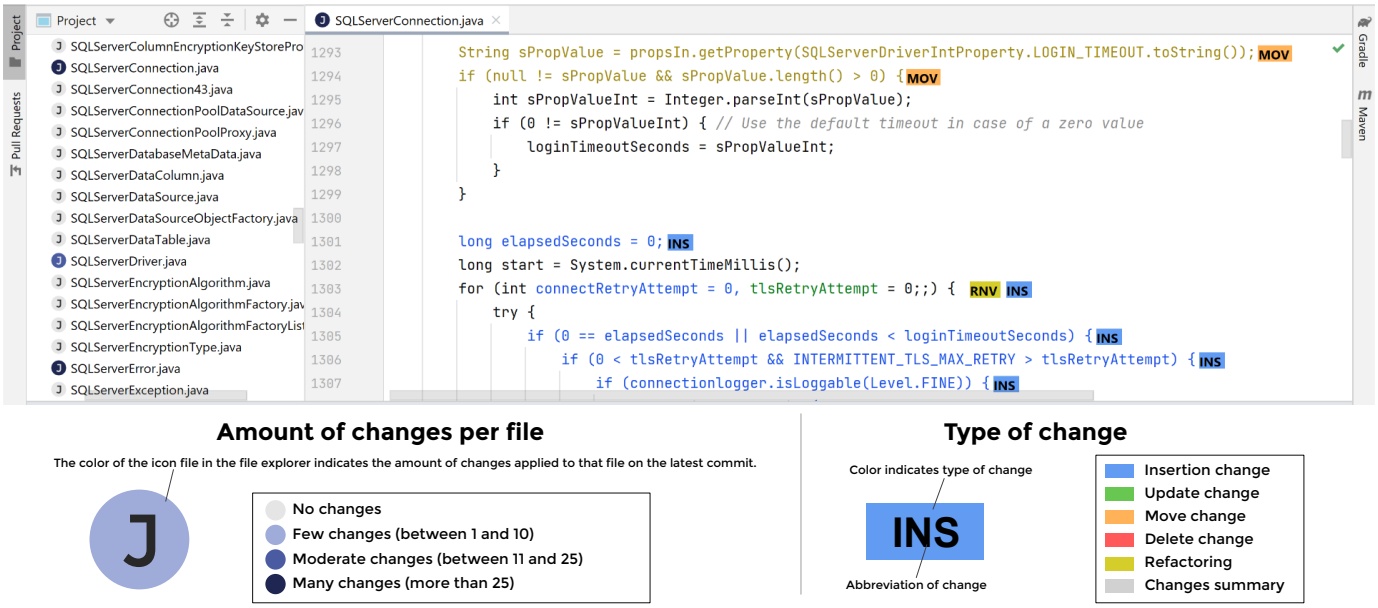


Fig. 2. Example for change-based source code highlighting showing modifications in `SQLSeverConnection.java` from the project *mssql-jdbc*. The left panel contains an enhanced file explorer: a color is assigned to each file icon according to the number of changes. The right panel shows the code editor with a highlighting that reflects the changes from the previous version. Font colors are assigned according to the kind of change, and visual markers placed at the end of each modified line give on-demand information about the change; texts inside the markers are acronyms of the kind of change.

ence information. Our proposed highlighting mechanism is accompanied by a number of visual icons that help developers obtain more information about a code change on demand. We developed a prototype that enhances two main components of *IntelliJ IDEA*, a popular Java editor. Figure 1 contrast our solution to existing ones, and Figure 2 gives an overview of the whole approach. Differently than *IntelliJ Editor*, Spike highlights different kinds of changes within a line, including additions, deletions, updates, and refactorings.

To assess the expressiveness and usability of our prototype, we conducted an exploratory user study comparing our approach with the *GitHub* code difference tool. We asked five professional software engineers to perform a code comprehension task involving large Git commits. Overall, engineers performed equally well using the GitHub code difference tool and our augmented editor. This indicates that our editor is able to assist complex comprehension tasks, although it offers significantly less exposed historical information.

## II. Related Work

***Visualizations in the code.*** Since *Seesoft* [4] has suggested visualizing software metrics as colored source code lines, diverse approaches have proposed adding different types of data into the code editor [5], for instance, showing data on code smells [6], call graphs [7], memory usage [8], [9], and software performance [10], [11]. In contrast, our work adds information about code changes to the editor. In the same direction, already *Seesoft* [4] as well as Harward *et al.* [12], [13] visualize information related to software history, for instance, source code age or developers that last edited a corresponding line. *I3* [14] adds change summaries as small timelines on method level to the code. However, we are not aware of an approach that uses visual augmentations to show fine-grained version difference information in the code editor.

***Visual code difference tools.*** Common source code difference tools compare two code versions and either show the information using a side-by-side or unified representation. Whereas, usually, changes are only detected and visualized at line level, only few tools visualize more fine-grained changes in a side-by-side view, i.e., which code sections were modified, added or deleted [15], [16], [17]. They also integrate this with timeline views [15] and architectural diagrams [16]. Differently from these tools, our approach only shows the current (newest) version of the code and highlights code changes in situ in the code editor based on a given previous version.

## III. Spike: Change-based Code Highlighting

Spike enhances two components of IntelliJ: the *File Browser* and the *Code Editor Panel*.

### A. File Browser

To easily spot which files contain changes, our first enhancement targets establishing a file-color icon based on the number of changes within a file. Figure 2 (left side) illustrates how the color icons are displayed in the file browser panel. The files are categorized based on a number of configurable thresholds. By default, our implementation considers four categories with the following thresholds: none – zero changes, few changes – between 1 and 10, moderate changes – between 11 and 25, and many changes – more than 25. We consider a change to be either an insertion, update, move, delete, or refactoring within a file. Note that there is a letter inside each

icon to indicate the file type. The letter is assigned by *IntelliJ*, for instance, *J* means Java file.

### B. Code Editor Panel

We enhance the code editor panel by highlighting recent code-change and adding visual markers to classify those changes.

***Color encoding.*** We support five change categories, each represented by a font color: insertion (), update (), move (), delete (), and refactoring (). As an illustration, Figure 2 shows a code section that involves three kinds of changes. Code sections without modifications remain with the default color font. The colors used by our tool are picked from the *IntelliJ IDEA* color palette. If users change the color palette, for instance, to a black background style, our tool also uses compatible colors. For instance, consider Figure 3, which shows a variety of source code changes done in the file `SQLServerConnection.java`. Spike indicates that one variable name was renamed and updated, two variables were updated, and a new variable was declared.

```
/* these static variables are used for the multithreaded tests */
private static final int NUM_OF_VERTEXES_PER_PARTITION = 20;     RNA
private static final int NUM_OF_EDGES_PER_VERTEX = 5;
private static final int NUM_OF_THREADS = 8; UPD
private static final int NUM_OF_PARTITIONS = 30; UPD
private static final int NUM_PARTITIONS_IN_MEMORY = 12; INS
```

Fig. 3. Example: variable renamig and constant updates

***Icons & popups.*** We append an icon to each modified line. Clicking on it provides details about the modification in a popup. Icon colors have the same color as the font, and inside each icon there is an abbreviated name of the type of change. Popup labels give information about the change and the commit that introduced the change: author username, email, and commit date. Figure 4 gives details about a renamed parameter refactoring, and it shows the previous name of the parameter. In case of deleted lines, since these lines are not in the current version (the code that developers are editing), we show a red icon where the code was deleted. Clicking on this icon reveals the deleted code.

### C. Spike Implementation

We implemented Spike as a plug-in for the *IntelliJ Java Editor*. We use IJM to detect the source code change at AST level [18], and *RefactoringMiner* to identify the refactorings [19], [20]. Our prototype works with *Git* repositories. It supports comparison of exactly two versions and, by default, compares the current version with the previous one; however, it also lets developers select an older version to compare with.

## IV. User Study

Five engineers from international companies volunteered to participated in our study. They had between 1 and 3 years

```
public void failsAfterMSorterSorting() {  RNM
  final String query = "select n_name fro
  Class<? extends Exception> typeOfExcept

  final long before = countAllocatedMemor
  final String controls = Controls.newBui
    .addException(ExternalSortBatch.class, ExternalSortBatch.INTERRUPTION_AFTER_SO
```

**RENAMED METHOD**
**Old method name:** testInterruptingAfterMSorterSorting
**Commit info**
**Author username:** Sudheesh Katkam
**Author email:** skatkam@maprtech.com
**Commit datetime:** 29/05/2015 19:50

Fig. 4. Popup example for renaming a method.

of professional experience in software development at outsourcing companies. All of them had at least one year of experience in professionally developing software in Java and using *IntelliJ IDEA*.

### A. Design

***Baseline.*** We used the *GitHub* code difference tool as the baseline for comparison. However, unlike our plugin, GitHub does not detect refactorings, which might introduce a bias in favor of our approach. To minimize this bias, we installed the *RefactoringMiner* plugin to the participants' Google Chrome browser. This Chrome extension highlights and summarizes refactorings within the GitHub code difference tool.

***Code under study.*** Based on a previous study [19], we employed a data set of commits that contains a diverse kind of refactorings and source code changes based on *RefactoringMiner*. We then selected two commits that contain a great variety not only of refactorings but also of line insertions, deletions, and updates, one from the *Apache Drill* project and the other from *Apache Giraph*.

***Task & procedure.*** Each participant was requested to analyze both commits and write a detailed commit message for them. We did not provide them the original commit message to avoid bias. Each commit was analyzed with a different tool: Spike and the baseline. To balance any learning effect, we randomized the commits and tools.

***Data collection.*** During the study, participants were asked to speak their thoughts and ideas out loud. After each commit, we requested them to fill two forms: i) the system usability scale form (SUS), and ii) the NASA Task Load Index (NASA-TLX). At the end of the session, we asked the participants to list advantages and disadvantages of our approach in comparison to the baseline.

The implementation of Spike and the commits under study are available online[1].

### B. Results

***Usability & cognitive load.*** Table I shows the SUS and NASA-TLX total scores for each participant. We measured the scores as advised in the original description of the SUS form [21] and NASA-TLX [22], respectively. *P3, P4, P5* rated the usability (SUS) of Spike with a better score than the GitHub difference tool; *P1* assigned a slightly better score to GitHub, and *P2* the

[1]https://github.com/testFooBar71/highlight-code-plugin

TABLE I
USABILITY AND COGNITIVE LOAD RESULTS

| | Session Time | SUS | | NASA TLX | |
|---|---|---|---|---|---|
| | | Spike | GitHub | Spike | GitHub |
| **P1** | 92 min | 75 | 80 | 45.3 | 38 |
| **P2** | 86 min | 57.5 | 57.5 | 45.7 | 42.3 |
| **P3** | 77 min | 90 | 70 | 13.7 | 45.3 |
| **P4** | 89 min | 70 | 47.5 | 60.3 | 65.7 |
| **P5** | 90 min | 77.5 | 57.5 | 39 | 62 |

same score to both. Regarding the task load index (NASA-TLX), *P3, P4, P5* reported a lower workload perception using our plugin in comparison to GitHub, and *P1, P2* reported a slightly higher workload perception using our plugin. All participants commented that Spike is useful and easy to understand. Overall, we observe that participants usability and workload perceptions of both tools are in a similar range for most participants, with two participants (*P3, P5*) ranking our plugin clearly better across both scales.

***File explorer.*** All participants first started expanding all folders in the file browser and identified based on the color encoding which files were changed. Since the projects under analysis contained many files, participants used the "Expand all" feature of *IntelliJ IDEA* in the file explorer. P1, P2, and P3 followed a top-to-bottom approach to navigate the file explorer. P5 followed a more random order. P4 preferred to close folders with no modified files; it shows that P4 easily distinguished files with modifications on the explorer. While, in some cases, participants missed files with few changes while scrolling, in most of the cases, all participants easily identified files with modifications, stopping the scroll when they detected a file with a different icon. P2 opened mostly files with many modifications. P4 and P5 suggested the possibility to filter only the files with modifications on the file explorer.

***Code highlighting.*** Except for P1, who had a slight confusion between blue from inserted code and white from not modified code, no other participant had issues with the font color encoding. Four participants did not comment on the color encoding specifically; they quickly associated the blue color with line insertions. However, P2 suggested using green for inserted code and blue for updated code. Despite our prototype had a minor bug in highlighting multiple inserted lines, all participants were able to describe the code changes done in all files and write a proper commit message.

***Icons.*** P1 and P3 had confusions with the DEL icon. They thought that the code on the line of the visual element was the deleted code. This might be because traditional code difference tools show deleted lines. But in our case, deleted lines are only shown on demand when users click the DEL icon. P2 mentioned that basic acronyms (INS, UPD, MOV, DEL) are easy to understand, but there are too many acronyms to remember; P2 suggested having a single acronym for all refactorings. However, P2 might not have realized remembering acronyms is not necessary, as the full name of the modification is shown on the popup.

***Popup.*** This feature probably had the most positive reception among all features. P2 and P5 said explicitly that the popup was useful to review the code before the modifications. P5 mentioned that the popup was useful to see who and when a change was made.

***General feedback.*** Participants used adjectives such as *good*, *useful*, and *interesting* to describe our plugin. P2 deemed the idea to see previous code without *git blame* seems useful. P3 said he liked Spike a lot, and he would use it. P5 highlighted the utility of showing refactorings.

***Advantages.*** P1, P3, and P4 highlighted the fact that you do not need to leave the IDE to see the changes. P2 said that "you don't even need to change the view within the IDE". P5 also highlighted the better precision of the plugin for moves or refactorings, which standard diff tools only detect as insertions or eliminations. P1 and P3 also noted the visual expressiveness of the tool as an advantage.

***Disadvantages.*** P1 mentioned that all the information shown can be heavy for the user. P2 and P3 mentioned that having many visual elements on the editor could be invasive. P4 pointed out not having the split diff view as the only disadvantage. P5 said that looking for modified files on the file explorer could take a long time, and that could be a disadvantage. P1 and P2 said that it is difficult to adapt to this new highlighting, and that they are used to the traditional highlighting.

***Recommendations for improvement.*** P3 and P4 suggested showing changes on the navigation sidebar of the editor. P4 and P5 suggested having the option to filter the modified files on the file explorer.

### C. Threats to Validity

***Code under study.*** The commits picked might not be representative or otherwise could have biased the results. To minimize this threat, we selected commits containing a variety of refactorings and code modifications of well-known software projects, based on the *RefactoringMiner* study data set [23]. We picked two commits with a comparable number of changes.

***Baseline.*** Although trying to find a comparable setup based on GitHub and the *RefactoringMiner* plugin for GitHub, this baseline might still introduce a bias. Participants might have preferred other available tools for the task.

***Tasks.*** The study only focused on analyzing commits, but not how well participants perceive the change information while writing code.

***Reliability of results.*** Our findings are based on feedback of five participant and their usability perceptions. Hence, quantitative results can only be considered preliminary, while qualitative findings might already be more reliable.

## V. Conclusion and Future Work

This paper proposes Spike as an unobtrusive extension of the source code editor to obtain fine-grained historical information. In an exploratory study, practitioners positively perceived our plugin and rated it comparable or better to using the *GitHub* code difference tool. Hence, our approach provides a new and useful option for analyzing code difference, one that fully integrates with the coding environment.

As future work, we plan to perform a controlled experiment to understand, in contrast to traditional code comparison tools, how and how well our prototype help developers to analyze source code changes.

## Acknowledgments

Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this article. Alexandre Bergel thanks ANID/FONDECYT (regular project 1200067). This work has also been funded by Deutsche Forschungsgemeinschaft (DFG) as part of research grant 288909335.

## References

[1] C. Hannebauer, M. Hesenius, and V. Gruhn, "Does Syntax Highlighting Help Programming Novices?" *Empirical Software Engineering*, vol. 23, no. 5, pp. 2795–2828, Oct. 2018.

[2] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software History Under the Lens: A Study on Why and How Developers Examine It," in *2015 IEEE International Conference on Software Maintenance and Evolution*. Bremen, Germany: IEEE, Sep. 2015, pp. 1–10.

[3] T. D. LaToza and B. A. Myers, "Developers Ask Reachability Questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1. Cape Town, South Africa: ACM Press, 2010, p. 185.

[4] S. Eick, J. Steffen, and E. Sumner, "Seesoft – A Tool for Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, Nov./1992.

[5] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual Augmentation of Source Code Editors: A Systematic Mapping Study," *Journal of Visual Languages & Computing*, vol. 49, pp. 46–59, Dec. 2018.

[6] E. Murphy-Hill and A. P. Black, "An Interactive Ambient Visualization for Code Smells," in *Proceedings of the 5th International Symposium on Software Visualization*. Salt Lake City, Utah, USA: ACM Press, 2010, p. 5.

[7] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. Santa Barbara, California, USA: ACM Press, 2011, p. 217.

[11] S. Baltes, O. Moseler, F. Beck, and S. Diehl, "Navigate, Understand, Communicate: How Developers Locate Performance Bugs," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Beijing, China: IEEE, Oct. 2015, pp. 1–10.

[8] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret, "Augmenting Static Source Views in Ides with Dynamic Metrics," in *2009 IEEE International Conference on Software Maintenance*. Edmonton, AB, Canada: IEEE, Sep. 2009, pp. 253–262.

[9] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz, "Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 579–591, May 2012.

[10] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code," in *2013 21st International Conference on Program Comprehension*. San Francisco, CA, USA: IEEE, May 2013, pp. 63–72.

[12] M. Harward, "CoderChrome: Augmenting Source Code with Software Metrics," Master's thesis, University of Canterbury, New Zealand, 2009.

[13] M. Harward, W. Irwin, and N. Churcher, "In Situ Software Visualisation," in *2010 21st Australian Software Engineering Conference*. Aukland, New Zealand: IEEE, 2010, pp. 171–180.

[14] F. Beck, B. Dit, J. Velasco-Madden, D. Weiskopf, and D. Poshyvanyk, "Rethinking User Interfaces for Feature Location," in *2015 IEEE 23rd International Conference on Program Comprehension*. Florence, Italy: IEEE, May 2015, pp. 151–162.

[15] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of Fine-Grained Code Change History," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. San Jose, CA, USA: IEEE, Sep. 2013, pp. 119–126.

[16] V. Uquillas Gómez, S. Ducasse, and T. D'Hondt, "Visually Characterizing Source Code Changes," *Science of Computer Programming*, vol. 98, pp. 376–393, Feb. 2015.

[17] V. Frick, C. Wedenig, and M. Pinzger, "DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts," in *2018 IEEE International Conference on Software Maintenance and Evolution*. Madrid: IEEE, Sep. 2018, pp. 705–709.

[18] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating Accurate and Compact Edit Scripts Using Tree Differencing," in *2018 IEEE International Conference on Software Maintenance and Evolution*. Madrid: IEEE, Sep. 2018, pp. 264–274.

[19] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and Efficient Refactoring Detection in Commit History," in *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 483–494.

[20] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, Mar. 2022.

[21] J. Brooke, "SUS: A quick and dirty usability scale," *Usability Evaluation in Industry*, vol. 189, 1996.

[22] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Human Mental Workload*, ser. Advances in Psychology, P. A. Hancock and N. Meshkati, Eds. North-Holland, 1988, vol. 52, pp. 139–183. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166411508623869

[23] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, p. 858–870.