ChangeViz: Enhancing the GitHub Pull Request Interface with Method Call Information

Lorenzo Gasparini,* Enrico Fregnan,† Larissa Braz,† Tobias Baum,‡ Alberto Bacchelli†
*Independent researcher, †University of Zurich, Switzerland, ‡Leibniz Universität Hannover, Germany

***Abstract*—Code review is a widely adopted software development practice aimed at finding defects, improving software quality, and transferring knowledge among developers. Performing an effective code review is a challenging task for developers. Two of the main challenges reviewers face are (1) understanding the content of a review change-set and (2) assessing the impact of a change on the codebase. Visualization techniques can be used to increase developers' understanding of a changeset to review and its context. However, only a few attempts have been made to apply visualization to code review.**

In this paper, we present a novel approach we devised to support developers in understanding GitHub pull requests. Our approach expands the GitHub interface with two lateral bars to let developers navigate to the definition/uses of the methods in the changeset under review.

We evaluated our approach's interface through (1) interviews with eight developers and (2) a survey with 12 participants. Based on the results of this evaluation, we implemented our approach in a web-based tool, ChangeViz.
Pre-print, data and materials, and demo video: https://doi.org/10.5281/zenodo.5175927.

***Index Terms*—Software Visualization, Code Review**

I. Introduction

Reviewing code written by other developers before integrating it into production is a widely used software development practice. It is aimed at finding defects [1], improving software quality [2], [3], and transferring knowledge among developers [4], [5]. Contemporary code review (*a.k.a.* Modern Code Review) is (1) change-based, (2) informal, (3) asynchronous, and (4) supported by tools [5]–[7].

Developers find it most challenging to (1) understand the changes they are set to review [4], [8], [9], especially when dealing with large sets of changes [1], and (2) assess the impact of a change [4]. Indeed, to conduct a review effectively, developers need not only to understand the code that was modified but also its context.

Researchers have proposed different approaches to support developers' understanding during code review: *e.g.*, reordering code changes highlighting their relations [3] and separating unrelated code changes [10].

Among the devised approaches, *visualizing interactive information on the code* might be a promising way to support developers during code review. Previous studies already showed the benefits of visualization techniques for developers [11]–[13]: *e.g.*, supporting them in better understanding or navigating the code. However, visualization techniques to support code review remain largely unexplored. In particular, using software visualization to support reviewing a single changeset (*e.g.*, a pull request) is largely unexplored. We argue that visualization techniques can help developers in understanding a changeset under review as well as assessing its impact on the whole codebase.

In this paper, we present ChangeViz, a publicly available [14] web-based tool we devised for GitHub pull requests. ChangeViz combines the interface offered by GitHub with the visualization approach of Stacksplorer [15]. Our tool adds two lateral bars (a *definitions* bar and a *references* bar) to the traditional GitHub interface, thus allowing the user to navigate from and to the method calls/declarations in the code under review. Through this functionality, we aim to support developers not only in understanding the content of a review change but also in better assessing its impact on the whole codebase. With ChangeViz, developers can quickly visualize the code related to the method calls/declarations without switching context from the review.

We evaluated a presentation of our visualization approach with developers, performing 8 interviews and a survey with 12 participants. We used the collected feedback to improve our visualization approach iteratively; then, we implemented it into ChangeViz. A video showing a demo of our tool is available here [14].

II. Background and Related Work

Software visualization is the use of visual means to study the structure and evolution of software [13]. In the context of code review, only a few approaches based on software visualization have been proposed so far.

Tymchuk et al. [16] devised ViDI, a tool to support developers in reviewing the design of a system. ViDI employs a city-based visualization paradigm, where classes are shown as buildings formed by the methods they contain. ViDI shows and focuses on the whole design of the system in which the code change is embedded. Oosterwal et al. [17] devised Operias, a tool that displays test coverage information concerning a single pull request under review.

Our approach employs the same visualization paradigm as Stacksplorer [15], but with the aim of reviewing GitHub pull requests. Stacksplorer works in the IDE and allows developers to navigate *upstream* from a method to the method callers or *downstream* to the method definition. Stacksplorer implements two lateral bars (on the right and left of the code window) that allow developers to navigate the method call relationships. This feature increases developers' awareness of the context of the code they are inspecting.

For this reason, we selected the visualization approach of Stacksplorer as an ideal candidate to be applied to code review. Being able to quickly inspect the declaration/use of a method used/declared in a changeset to review without interrupting the flow of the review has the potential to support developers' understanding of the changeset and its impact on the codebase.

Furthermore, an investigation conducted by Krämer *et al.* [18] showed the effectiveness of Stacksplorer in supporting developers in performing code maintenance, decreasing the time needed to complete these tasks. This result gives an initial indication of the benefits of such a visualization paradigm and further contributed to our decision to apply this approach in the context of code review. Adapting the paradigm of Stacksplorer to GitHub pull requests came with a set of challenges (further explained in Section VI): *e.g.*, (1), differently from the IDE, a PR contains both new and old code; and (2) a PR may contain different programming languages.

III. Preliminary Evaluation

Before implementing our tool, we evaluated the visualization we devised. To this aim, we interviewed eight software developers with experience in Java and the GitHub review interface. We walked them through a storyboard portraying the interface of a ChangeViz's prototype and a sequence of realistic interactions with it. We collected participants' feedback by using a think-aloud protocol [19] during the walk-through.

After each interview, we analyzed the participants' comments to identify problems and areas of improvement, if any, then modified the prototype accordingly. We performed a total of three iterations, after which we reached a stable version that did not exhibit any design flaws according to the participants.

Most of the participants appreciated how ChangeViz is similar to the interface of the code review tools they are familiar with. For instance, one of the participants explained: "[ChangeViz] is nice because developers are familiar with this view." In the second iteration of our evaluation, following a participant's suggestion, we modified the call-graph exploration feature so that when the class containing the reference/definition is opened, the code is automatically scrolled to the position of the reference/definition and an arrow indicates the exact location.

The original design of the tool also included a horizontal scroll bar to visualize two tabs next to each other. However, this feature received criticism from multiple participants in the third iteration; for example, a participant explained: "The horizontal bar scrolling is more fancy than useful, you could just select the tab with a click." Therefore, we removed it from our design.

Once we reached a stable version of the design, we evaluated how developers would assess their usability by seeing a mock in action in a video. We did this through a survey with 12 participants (we also invited the participants who took part in the interviews). All participants had at least two years of programming experience, most of them reported to program daily and to review code on a weekly basis, and they mainly used GitHub pull requests for reviews. As with the interviews, our goal was to collect feedback on the presentation of our visualization approach before implementing it in a tool.

In the survey, we asked participants to watch a short (less than five minutes) video explaining the tool's interface and the available interactions. Then, we asked them to evaluate the perceived usability of the tool through an adapted SUS (System Usability Scale [20]) questionnaire: We removed the last two items (which refers to the use of the tool in practice) since participants could not try the tool. In this way, we obtained a score in a range from 0 to 80. Then, to compare this score with the finding of existing literature, we normalized it from 0 to 100. Overall, our design received a usability score of 66.67 (positioning between an 'OK' and 'Good' design [21]). In particular, participants believed the tool would be easy to learn and use (avg. item score of 4 and 3.5, respectively).

IV. ChangeViz

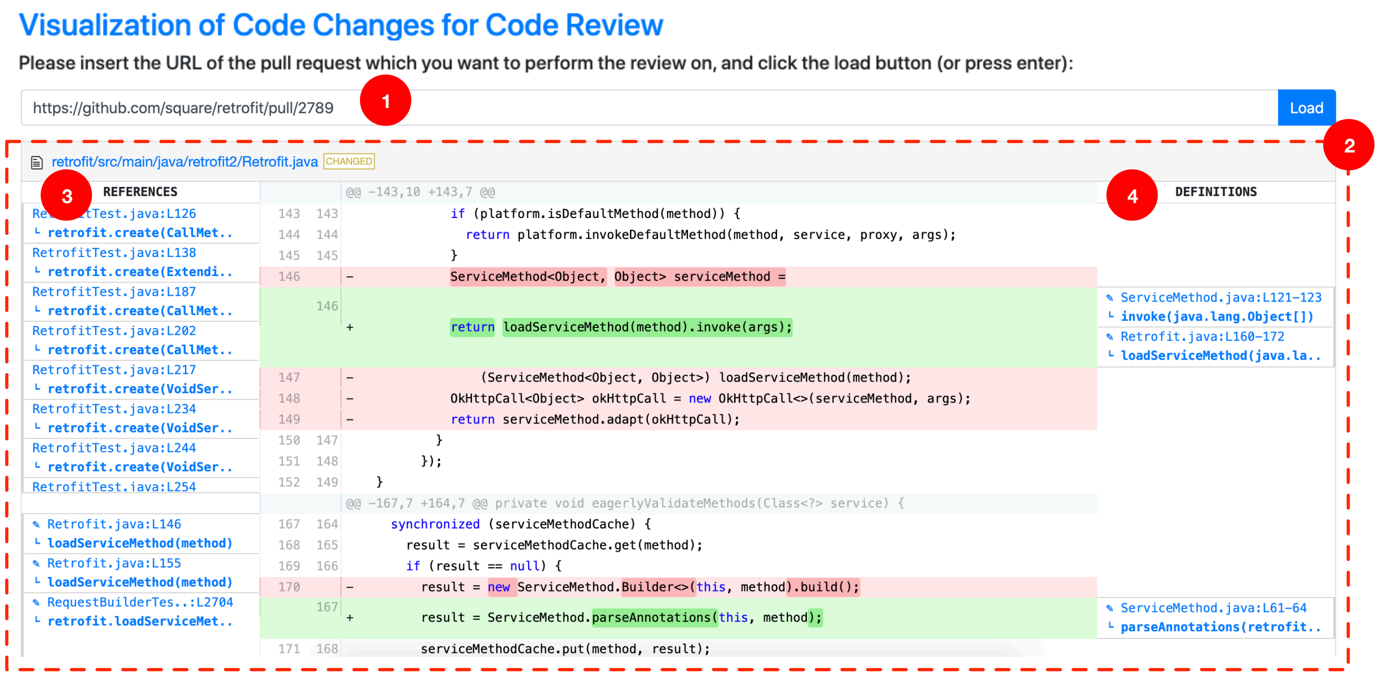
Based on the positive feedback collected by developers, we implement our visualization approach in a tool, called ChangeViz.

Figure 1 shows the main view of ChangeViz. In this example, we use our tool to visualize pull request #2789 in Retrofit [22]. The item marked with a ① in Figure 1 points to the *loading* bar: The user inputs the pull request URL and clicks on the *Load* button to perform a review. The tool then loads the changeset diff right below ②, similarly to the diff proposed by standard code review tools, including GitHub.

ChangeViz presents the unified diffs of each modified file one after the other and applies syntax highlighting. To expand the context of a diff, the user can click on the path of a modified file. The tool will show the complete diff of the selected file with an infinite context.

ChangeViz presents two sidebars (*i.e.*, *References* and *Definitions*) that allow the reviewer to navigate the context of the change under review by following the call/declaration relationships of the methods. These lateral bars represent the main novelty of ChangeViz's interface with respect to GitHub's review interface. Figure 2 details the sidebars.

References sidebar. The *References* sidebar is positioned on the left-hand side of the code (③ in Figure 1). Figure 2a shows how ChangeViz reports the reference information. Figure 2b shows an example of the information displayed to the user when their mouse hovers over an element in the references' bar. For each method whose declaration is contained in the portion of code shown in the diff, ChangeViz automatically displays the list of references (*i.e.*, calls) to the method in other parts of the codebase. More specifically, it displays the following information about each reference to a method: (1) the Java file containing the method call, (2) the line number where it is located, and (3) the method call itself (Figure 2a). In addition, if the file containing the method call was also modified in the changeset under review, the tool shows a pencil next to the file name to inform the user.


Visualization of Code Changes for Code Review
Please insert the URL of the pull request which you want to perform the review on, and click the load button (or press enter):
https://github.com/square/retrofit/pull/2789
Load
1
2
retrofit/src/main/java/retrofit2/Retrofit.java
CHANGED
3
REFERENCES
4
DEFINITIONS

References:

```
RetrofitTest.java:L126
└ retrofit.create(CallMet..
RetrofitTest.java:L138
└ retrofit.create(Extendi..
RetrofitTest.java:L187
└ retrofit.create(CallMet..
RetrofitTest.java:L202
└ retrofit.create(CallMet..
RetrofitTest.java:L217
└ retrofit.create(VoidSer..
RetrofitTest.java:L234
└ retrofit.create(VoidSer..
RetrofitTest.java:L244
└ retrofit.create(VoidSer..
RetrofitTest.java:L254

✎ Retrofit.java:L146
└ loadServiceMethod(method)
✎ Retrofit.java:L155
└ loadServiceMethod(method)
✎ RequestBuilderTes..:L2704
└ retrofit.loadServiceMet..
```

```
@@ -143,10 +143,7 @@
              if (platform.isDefaultMethod(method)) {
                return platform.invokeDefaultMethod(method, service, proxy, args);
              }
-             ServiceMethod<Object, Object> serviceMethod =
+             return loadServiceMethod(method).invoke(args);
-                 (ServiceMethod<Object, Object>) loadServiceMethod(method);
-             OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args);
-             return serviceMethod.adapt(okHttpCall);
            }
          });
    }
@@ -167,7 +164,7 @@ private void eagerlyValidateMethods(Class<?> service) {
      synchronized (serviceMethodCache) {
        result = serviceMethodCache.get(method);
        if (result == null) {
-         result = new ServiceMethod.Builder<>(this, method).build();
+         result = ServiceMethod.parseAnnotations(this, method);
          serviceMethodCache.put(method, result);
```

Definitions:

```
✎ ServiceMethod.java:L121-123
└ invoke(java.lang.Object[])
✎ Retrofit.java:L160-172
└ loadServiceMethod(java.la..
✎ ServiceMethod.java:L61-64
└ parseAnnotations(retrofit..
```

Fig. 1: Interface of CHANGEVIZ visualizing a pull request from GitHub.

Each list of references is vertically aligned with the corresponding (portion of a) method declaration, establishing a visual connection between the two, and a scroll bar can be used to browse if needed. Moreover, the tool shows links to access the source code of methods called by the changed code.

When a specific method call is hovered with the mouse cursor, an information box appears (Figure 2b) and provides the file's path containing the reference and the complete method call. By clicking on the method call, the user can access the source code of the file that contains it (displayed in a modal window). The modal window's content varies depending on whether the file was modified by the changeset under review: If the file was modified, the modal window shows the unified diff of the file; otherwise, it shows its plain source code only. In both cases, the code is syntax-highlighted. Moreover, the file is displayed directly to the position of the method call, whose line number is highlighted to ease its location. Finally, a user can close the modal window by clicking outside its boundaries or selecting the close button on the window's top right corner.

Definitions sidebar: The *Definitions* sidebar is displayed on the right-hand side of the pull request ((4) in Figure 1). More details are offered in Figure 2c and Figure 2d: The former shows the definition details offered by CHANGEVIZ, while the latter shows an example of the information displayed to the user when the mouse hovers on an element in the definitions bar.

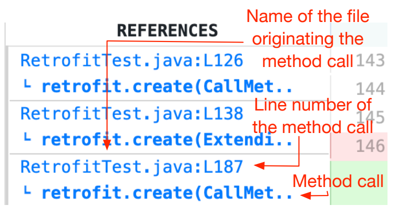
The *definitions* sidebar mimics the 'Go to Declaration' feature offered by common IDEs. It allows the reviewer to obtain information about the definition of the methods invoked within the diff (methods declared in external libraries are excluded). For each method call in the diff, the *Definitions* sidebar displays the following information: (1) the name of the file containing the called method's definition, (2) the line range of the definition within that file, and (3) the signature of the called method (as shown in Figure 2c). In the same fashion as with the references sidebar, a pencil next to the filename indicates whether the file containing the method definition was modified in the current pull request. The definition information is vertically positioned on the same line as the corresponding method invocation, which is expanded when it contains multiple method calls.

By hovering on an element in the definitions sidebar, the user can obtain: (1) the path of the file containing the method definition and (2) the qualified signature (which includes the package and the class to which it belongs) of the method (Figure 2d). Concerning the references, clicking on a definition opens the source code (or diff, if it was modified) of the file containing the method's definition. The code is positioned at the beginning of the method declaration, and the corresponding line range is highlighted.

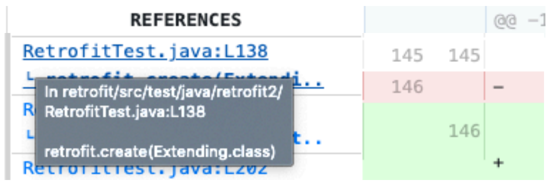
Our video [14] offers an overview of how our tool could support reviewers during their tasks.

V. CHANGEVIZ: IMPLEMENTATION DETAILS

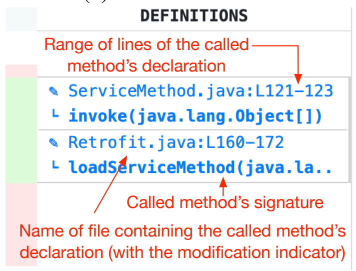
CHANGEVIZ uses a database to cache the computed method call information. To compute the references and definitions information, it retrieves the pull request's diff from GitHub and clones the entire Git repository locally. The diff is then applied to obtain the codebase's state after the application of the pull request, and the method call extraction engine is invoked to extract the method call information of each modified Java file.


REFERENCES
Name of the file originating the method call
RetrofitTest.java:L126
retrofit.create(CallMet..
RetrofitTest.java:L138
Line number of the method call
retrofit.create(Extendi..
RetrofitTest.java:L187
Method call
retrofit.create(CallMet..
143
144
145
146

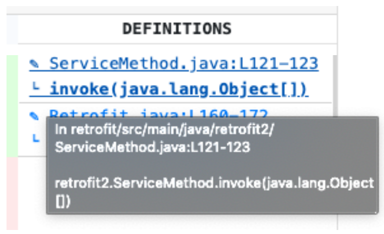
(a) References detail



(b) Mouseover information box of references details


DEFINITIONS
Range of lines of the called method's declaration
ServiceMethod.java:L121-123
invoke(java.lang.Object[])
Retrofit.java:L160-172
loadServiceMethod(java.la..
Called method's signature
Name of file containing the called method's declaration (with the modification indicator)

(c) Definitions details


DEFINITIONS
ServiceMethod.java:L121-123
invoke(java.lang.Object[])
Retrofit.java:L160-172
In retrofit/src/main/java/retrofit2/
ServiceMethod.java:L121-123
retrofit2.ServiceMethod.invoke(java.lang.Object[])

(d) Mouseover information box of definitions details

Fig. 2: Details of CHANGEVIZ's interface.

The engine is a Java program that uses the JavaParser and JavaSymbolSolver[1] libraries to perform the static analysis of a Java code base and extract the references and definitions of a given set of files. When the extraction is completed, the computed method calls are cached in the database and sent to the front-end, which, in turn, visualizes them in the sidebars.

When the user clicks on a modified file to expand its context, a reference or a definition, the tool restores the Git repository's state following the application of the pull request. Then the tool extracts either the source code or the full diff (infinite context) of the requested file, which the user can visualize in the modal window.

VI. Design Decisions and Limitations

While building CHANGEVIZ, we took some design decisions to guide our implementation that might constitute limitations to be addressed in future investigations.

Emphasis on the new code chunk. A change in GitHub is composed of two code chunks: (1) the original code before the change was introduced (*old chunk*) and (2) the code after it was applied (*new chunk*). CHANGEVIZ displays information only for the method declarations and uses contained in the new code chunk. We decided to display only the method call information related to the new code chunk (disregarding the one of the old chunk) to avoid information overload. Displaying method information on both the old and new code chunk may unnecessarily increase the cognitive load on the reviewer. Moreover, we argue that the new code chunk is more relevant for reviewers compared to the old one because reviewers need to assess how the added code modifies the existing system.

Nonetheless, the choice of focusing on the new code chunk might have an impact on the potential benefits offered by CHANGEVIZ. Further studies are necessary to understand whether and how call information on the old code chunk is relevant and how this can be effectively combined with the information already visualized by our tool.

External API methods. When a call in the diff references a method in an external library, CHANGEVIZ cannot inspect its definition. This limitation comes from the variety of build systems used by Java projects, which complicates the automated retrieval and resolution of external libraries. However, given the widespread usage of external libraries, including this functionality in CHANGEVIZ may be a valuable addition to further increase the support our tool offers.

Navigation. On the one hand, CHANGEVIZ does not allow the user to iteratively explore the context of references and definitions. We excluded this feature from CHANGEVIZ as it might add a significant cognitive load on the reviewers. Indeed, the results obtained by Kramer *et al.* [18] suggest that navigating along multiple edges of the call graph at once may pose a cognitive challenge for developers. On the other hand, if the list of callers or callees grows longer, it becomes cluttered. Therefore future research should be designed and carried out to investigate aggregation and interaction methods to facilitate visual exploration of the references, as their number grows.

CHANGEVIZ as a stand-alone application. We implemented CHANGEVIZ as a stand-alone web application. Developers need to navigate to the tool's URL to use it to review their GitHub pull requests. This choice may limit the future use of CHANGEVIZ: The majority of developers do not like to use tools that disrupt their workflow (*e.g.*, tools that are not integrated with the tools they already use) [23]. In this vein, a future iteration of CHANGEVIZ could be offered as a browser extension that integrates with the GitHub review environment, to which most developers are already accustomed.

Focus on Java and scalability. CHANGEVIZ relies on Java-

[1]JavaParser and JavaSymbolSolver: https://javaparser.org

Parser and JavaSymbolSolver to analyze the content of a pull request and to extract the method calls and declarations. For this reason, our tool can only display references/definitions information for the Java files in a pull request, thus disregarding files written in other programming languages. Since our tool relies on a parser to extract the information to visualize, in our implementation, we restricted our focus to one programming language. Extending the support of the tool to multiple programming languages increases its applicability. Therefore, we plan to perform further investigations on the impact of adding other programming languages to CHANGEVIZ. JavaParser needs to analyze the whole code snapshot of a pull request to extract information on method calls/declarations. This task can not be done before the user specifies the pull request to review and might result in a longer waiting time before CHANGEVIZ is ready for the review. Moreover, since JavaParser only analyzes static code snapshots, it can not extract run-time information: *e.g.*, dynamic binding.

VII. Conclusion and Future Work

In this paper, we presented a visualization approach to support developers in reviewing a GitHub pull request. Our visualization approach enhances the way in which pull requests are displayed on GitHub. Following a visualization approach similar to STACKSPLORER [15], our approach complements the GitHub interface with two lateral bars, a *References* sidebar, and a *Definitions* sidebar. These allow the user to navigate to the code defining or using the methods contained in a pull request. We evaluated our approach with developers using (1) a set of eight interviews and (2) an online survey with 12 participants. Based on the collected feedback, we improved our approach and implemented it in a tool (CHANGEVIZ).

To the best of our knowledge, CHANGEVIZ constitutes one of the first attempts at devising a visualization approach to support developers during code review. Our tool tackles two of the main challenges developers face during code review [4], [8]: (1) understanding of the content of a review changeset and (2) assessing the impact of a change on the existing codebase. Nonetheless, CHANGEVIZ is still an early prototype with shortcomings that need to be addressed. For example, our tool is currently designed as a stand-alone application, despite the fact that developers dislike tools that are not well-integrated into their existing workflow. This might limit the adoption of our tool in practice. For this reason, we envision transforming our tool from a stand-alone application to, for instance, a GitHub plug-in.

Acknowledgment

A. Bacchelli, L. Braz, and E. Fregnan acknowledge the support of the Swiss National Science Foundation through the SNSF Projects No. PP00P2_170529 and PZ00P2_186090.

References

[1] T. Baum and K. Schneider, "On the need for a new generation of code review tools," in *Proceedings of the International Conference on Product-Focused Software Process Improvement*, 2016, pp. 301–308.

[2] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: an effective verification process," *IEEE Software*, vol. 6, 1989.

[3] T. Baum, K. Schneider, and A. Bacchelli, "On the optimal order of reading source code changes for review," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2017, pp. 329–340.

[4] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 712–721.

[5] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[6] P. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.

[7] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A faceted classification scheme for change-based industrial code review processes," in *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 74–85.

[8] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.

[9] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[10] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 341–350.

[11] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola, "Code Park: A New 3D Code Visualization Tool," in *Proceedings of the Working Conference on Software Visualization*, 2017.

[12] S. G. Eick, J. L. Steffen, and E. E. Sumner, "Seesoft—A Tool for Visualizing Line Oriented Software Statistics," *Transactions on Software Engineering*, 1992.

[13] A.-L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väätäjä, "Software visualization today: Systematic literature review," in *Proceedings of the International Academic Mindtrek Conference*, 2016, pp. 262–271.

[14] "CHANGEVIZ materials," https://doi.org/10.5281/zenodo.5175927.

[15] 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 symposium on User interface software and technology*, 2011, pp. 217–224.

[16] Y. Tymchuk, A. Mocci, and M. Lanza, "Code review: Veni, vidi, vici," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 151–160.

[17] S. Oosterwaal, A. v. Deursen, R. Coelho, A. A. Sawant, and A. Bacchelli, "Visualizing code and coverage changes for code review," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 1038–1041.

[18] J.-P. Krämer, T. Karrer, J. Kurz, M. Wittenhagen, and J. Borchers, "How tools in ides shape developers' navigation behavior," in *Proceedings of the Conference on Human Factors in Computing Systems*, 2013, pp. 3073–3082.

[19] A. H. Jørgensen, "Thinking-aloud in user interface design: a method promoting cognitive ergonomics," *Ergonomics*, vol. 33, no. 4, pp. 501–507, 1990.

[20] J. Brooke *et al.*, "Sus-a quick and dirty usability scale. usability evaluation in industry," *Usability evaluation in industry*, vol. 189(194), pp. 4–7, 1996.

[21] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," *Intl. Journal of Human–Computer Interaction*, vol. 24, no. 6, pp. 574–594, 2008.

[22] "Square retrofit pull request #2789 on GitHub," https://github.com/square/retrofit/pull/2789, 2018.

[23] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 672–681.