The Journal of Systems & Software 195 (2023) 111506

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Graph-based visualization of merge requests for code review☆

Enrico Fregnan *, Josua Fröhlich, Davide Spadini, Alberto Bacchelli

University of Zurich, Zurich, Switzerland

ARTICLE INFO

Article history:
Received 19 April 2021
Received in revised form 16 June 2022
Accepted 3 September 2022
Available online 12 September 2022

Dataset link: https://doi.org/10.5281/zenodo.7047993

Keywords:
Modern code review
Software visualization
Empirical software engineering

ABSTRACT

Code review is a software development practice aimed at assessing code quality, finding defects, and sharing knowledge among developers. Despite its wide adoption, code review is a challenging task for developers, who often struggle to understand the content of a review change-set. Visualization techniques represent a promising approach to support reviewers. In this paper we present a new visualization approach that displays classes and methods in review changes as nodes in a graph. Then, we implemented our graph-based approach in a tool (*ReviewVis*) and performed a two-step feedback collection phase to assess the developers' perceptions on the tool's benefits through (1) an in-company study with nine professional software developers and (2) an online survey with 37 participants. Given the positive results obtained by this first evaluation, we performed a second survey with 31 participants with a specific focus on supporting developers' understanding of a review change-set.

The collected feedback showed that the developers indeed perceive that *ReviewVis* can help them navigate and understand the changes under review. The results achieved also indicate possible future paths to use software visualization for code review.

Data and Materials: https://doi.org/10.5281/zenodo.7047993

1. Introduction

Code review is a software development practice aimed at improving software quality (Ackerman et al., 1989; Baum et al., 2017), finding defects (Baum and Schneider, 2016), and sharing knowledge among developers (Bacchelli and Bird, 2013). Code review was originally formalized by Fagan as a highly structured process in the form of code inspections (Fagan, 1976). Despite its benefits, formal code inspections proved to be cumbersome and time-consuming when applied in practice (Shull and Seaman, 2008). For this reason, most software companies and projects adopt a lighter-weight version of code inspections, often referred to as Modern Code Review (MCR) (Rigby and Bird, 2013). Modern Code Review is (1) informal, (2) asynchronous, (3) supported by tools, and (4) change-based (Rigby and Bird, 2013; Sadowski et al., 2018; Baum et al., 2017, 2016b).

Despite the evolution of code review tools in the last decade, reviewing code is still perceived by developers as a challenging task (MacLeod et al., 2017). In particular, developers report challenges when trying to understand the content of change sets to review (Tao et al., 2012; Bacchelli and Bird, 2013; MacLeod et al., 2017), especially when dealing with large code changes (Baum and Schneider, 2016). Indeed, previous studies confirmed, by interviewing (Bacchelli and Bird, 2013; Baum et al., 2016a) and surveying (Tao et al., 2012) developers as well as by analyzing code review discussions (Pascarella et al., 2018), that understanding is a main challenge of code review.

To address the issues created by the difficulty in understanding the change set under review, researchers have proposed a number of approaches. For example researchers proposed to help reviewers reduce their mental load and improve their review performance by ordering code changes in a specific flow (Baum et al., 2017), by decomposing unrelated changes into separate code reviews (Barnett et al., 2015), and by providing support for searching systematic changes and detecting anomalies (Zhang et al., 2015).

Another promising approach to support developers during code review tasks is *visualizing interactive information on the code*. Indeed, previous studies already showed that visualization techniques can help developers in better understanding, navigate, and maintain code (Khaloo et al., 2017; Eick et al., 1992; Bragdon et al., 2010; Mattila et al., 2016; Bedu et al., 2019). Tymchuk et al. made a first attempt to bring interactive visualization to support the review of an entire system (as opposed to a code change), by devising a tool (*ViDI*) to visualize the design of the system using a city-based visualization paradigm (Tymchuk et al., 2015). Gasparini et al. (2021) proposed a tool (ChangeViz) to instead complement an existing review interface (specifically, the GitHub interface) with two lateral bars to display information about

☆ Editor: Shane McIntosh.

* Corresponding author.
E-mail addresses: fregnan@ifi.uzh.ch (E. Fregnan), josua.froehlich@uzh.ch (J. Fröhlich), dspadini@fb.com (D. Spadini), bacchelli@ifi.uzh.ch (A. Bacchelli).

https://doi.org/10.1016/j.jss.2022.111506

method calls contained in the changes to inspect. Yet, interactive visualizations remain largely unexplored at code review time and it is unclear whether and how they can be beneficial.

In this paper, we proceed on this line of research. We develop a visualization approach for code review changes to support developers' understanding of review change-set and conduct a study to investigate developers' perception on it. We study whether and when developers find such a visualization most useful, what the perceived benefits and problems are, and—overall—whether they anticipate it could help them perform better code reviews. In particular, we asked developers if they perceived the devised approach would increase their understanding of a review change-set. Differently from Tymchuk et al. (2015), our approach supports the review of a single review change-set instead of a review of the whole system. The most widespread form of code review is change-based, rather than system-based (Rigby and Bird, 2013; Sadowski et al., 2018; Baum et al., 2017, 2016b). Moreover, we move from a city-based paradigm to a different visualization approach (an interactive graph), more suitable to show the links between code entities.

Displaying relations among classes and methods is beneficial to support developers' understanding of the code (Bragdon et al., 2010; D'Ambros et al., 2009; Hanakawa, 2007). To display this information, we adopt a graph-based visualization approach because graphs are recognized as ideal for presenting relations among data (Munzner, 2014). Compared to, for instance, a city-based paradigm (as the one proposed by Tymchuk et al. (2015)), we argue that graph-based visualization constitutes a better solution to show connections between entities (*e.g.*, classes and methods) in a review change-set. City-based visualization tools that display relationships (*e.g.*, caller-callee relationships) among software entities might overwhelm the user with too much information, generating overly complex city models (Jeffery, 2019).

Similarly to ChangeViz (Gasparini et al., 2021), our graph-based visualization approach is aimed at improving developers' understanding of a review change-set. However, while ChangeViz focuses on allowing developers to navigate through method calls/declarations, our graph-based approach gives reviewers an overview of a merge request and how its entities are connected.

Our visualization approach displays classes and methods as nodes of a graph and the structural coupling relations among them as links. The graph shows if a node was added, deleted, or changed in the current merge request. Entities related to the ones changed in the current code review (but not included in it) can also be visualized in the graph. We implemented our visualization approach in a tool, called *ReviewVis*, which works for merge requests in GitLab (*i.e.*, the code review unit of GitLab, which is similar to pull-requests in GitHub (Gousios et al., 2014)).

Subsequently, we collected feedback on *ReviewVis* performing (1) a study with nine professional developers from the same software development company, who used our visualization approach for two weeks, and (2) two online surveys with, respectively, 37 and 31 participants, who watched online videos of the main features offered by our visualization. In the first part of our evaluation (in-company study and first survey), we focused on investigating the general perceived benefits and applicability of *ReviewVis*. Given the positive results collected in this phase, we performed a second survey to understand the perceived impact of the devised tool on developers' understanding of a review change-set.

In the in-company study, developers positively assessed the support provided by *ReviewVis*, *i.e.*, to understand the structure of the merge request under review. These results were confirmed by the participants of the online surveys. Overall, our findings confirmed the potential of a graph-based visualization technique as a means to support code review and suggest promising future investigations in this line of research. Participants in the in-company study implicitly compared the use of *ReviewVis* (together with Gitlab) to the use of only Gitlab during code review. In fact, these participants use standard Gitlab for doing code review in their daily practice. Participants did not report any significant difficulty with the tool. Although this provides us with initial indications that the tool is usable and does not cause information overload, further studies should be carried out to evaluate, *e.g.*, whether this visualization leads to information overload and its effects on code review performance.

Participants perceived our tool as suited to support them in reviewing medium-sized changes (4–7 files). Instead, they perceived the tool as unnecessary for small changes (1–3 files) because these changes are deemed easier to understand by themselves. The participants also did not consider that the tool might be beneficial for large changes (more than 7 files) because its current approach limits its benefits in these large reviews. These findings underline the importance of further work on this area to better support these cases.

The contributions of this paper are the following:

A graph-based visualization for code review. A graph-based visualization approach, where each node represents a class or a method. The links in the graph represent instead dependencies among software entities (nodes in the graph): *e.g.*, method calls.

An implementation of the devised visualization. A tool (*ReviewVis*), created to interactively visualize the content of a GitLab merge request for review using our graph-based visualization approach. *ReviewVis* is available in our replication package (Fregnan et al., 2020).

An evaluation of the devised visualization based on developers' perceptions. We collected developers' feedback on the benefits of our tool (in particular on developers' understanding of a review change-set) through an *in-company study* and two *online surveys*. Developers involved in the in-company study reported the benefits they perceived using a graph-based visualization tool to perform code review. In particular, developers reported that they felt the devised tool helped them navigate and understand the code. These findings were corroborated by participants of both online surveys.

Structure of the paper. In Section 2, we present previous studies related to this paper. Section 3 describes our visualization approach and the design of the tool. In particular, it focuses on the requirements we identified and the implementation choices we took to build *ReviewVis*. Section 4 reports the methodology and results of our in-company and online evaluation of *ReviewVis*. Section 5 discusses the implications of our findings for researchers and practitioners, while Section 6 reports the limitations of this study. Finally, in Section 7, we summarize the main results of this study.

2. Background

2.1. Code visualization techniques

Software visualization has been defined as the use of visual means to study the structure, behavior, and evolution of software (Mattila et al., 2016; Diehl, 2007). In recent years, researchers focused on developing visualization techniques to support developers in understanding code changes (Gómez et al., 2015; D'Ambros et al., 2010). Approaches such as *Commit 2.0* (D'Ambros et al., 2010) highlight the packages, classes, and methods that have been modified and provide developers with code metrics (*e.g.*, LOC). However, only a few attempts have been made

to integrate visualization techniques into code review (Tymchuk et al., 2015). Tymchuk et al. (2015) devised a tool (ViDI) that employs static code analysis to guide the review of the design of the entities in an *entire software system*. ViDI is a city-based code visualization: Classes act as bases on which methods are stacked, forming the visual equivalent of a building. Wettel et al. (2011) previously assessed the benefits of this kind of software visualization to improve developers' code understanding.

As opposed to our tool, ViDI focuses on the review of the whole system and not on the review of a specific merge request. Moreover, a graph-based approach allows developers to immediately understand the relationships between software entities, represented as links in the graph. On the contrary, to the best of our knowledge, attempts to display relationships among code entities using a city-based visualization paradigm have so far not been successful. As highlighted by Jeffery, including relationships in city-based visualization approaches might lead to the creation of overly complex models, which overwhelm developers with excessive information (Jeffery, 2019).

Gasparini et al. (2021) focused on how to support developers when reviewing a single review change-set (*i.e.*, a GitHub pull request). To this aim, they developed a tool called ChangeViz that complements the existing GitHub pull requests interface with two lateral bars. These bars display (1) the definitions of the methods used in the review change-set and (2) the usages of the methods declared in the change-set under review. Both ChangeViz and our graph-based approach focus on supporting developers' understanding of a review change-set. As opposed to our approach, the work of Gasparini et al. (2021) does not provide developers with a summarizing visualization of the structural relationships among the entities under review.

A different approach was proposed by Oosterwaal et al. (2016). The authors developed a tool, called Operias, to provide visual information about test cases to support reviewers. Although both Operias and *ReviewVis* focus on the single review change-set (Pull Requests and merge requests, respectively), their aim is different: The former complements the displayed review changes with information on the related tests, while the latter gives reviewers an overview of how the entities in the review change-set are connected to each other.

2.2. Supporting developers during code review

Over the years, a vast amount of approaches have been developed to support developers during code review. Some approaches focused on improving the way changes are displayed to the reviewers: *e.g.*, re-ordering review changes (Baum et al., 2017, 2019) or untangling unrelated changes contained in the same Pull Request (Wang et al., 2019; Tao and Kim, 2015; Dias et al., 2015). The former category of approaches aims at reducing developers' cognitive load while performing code review to increase their performance. To this aim, Baum et al. (2017) developed a theory to create a similarity-based review changes order as opposed to the alphabetical order, currently used in widely adopted code review tools (*e.g.*, Gerrit[1]).

The order in which changes are presented for review is not the only factor that can impact reviewers' understanding and performance: Often, review change-sets contain changes tackling different issues, making their review harder for developers. Such change-sets are not a rare occurrence (Tao and Kim, 2015) and they constitute a significant challenge for reviewers (Tao et al., 2012). To tackle this issue, researchers proposed a variety of techniques to divide large review change-sets into smaller sets containing only related code changes (Wang et al., 2019; Dias et al., 2015; Barnett et al., 2015). These approaches exploit different relations among entities in the code to cluster related changes: *e.g.*, Barnett et al. proposed ClusterChanges, a tool that relies on *def-use* relations to decompose cluster changes (Barnett et al., 2015), while Wang et al. exploited also links between methods and classes (*e.g.*, method overriding or class inheritance) to develop an automatic change untangling approach called CoRA (Code Review Assistant) (Wang et al., 2019).

Finally, some approaches aim at guiding developers during code review. Checklist-based reading is a common guidance approach used in the context of code review (Rong et al., 2012; McConnell, 2004): A checklist tells reviewers what they should focus on while reviewing the code. MacLeod et al. investigated how developers perform code review at Microsoft and defined as best practice for reviewers the creation and use of a project-specific review checklist (MacLeod et al., 2017). Developers can also be told explicitly *how* to review through the use of review *strategies*. The use of strategies to guide developers has already been proven effective in the context of test-driven development (LaToza et al., 2020) and debugging (Francel and Rugaber, 2001), while their application to code review is still object of investigation (Gonçalves et al., 2020).

3. Approach

To support developers' understanding of code changes during code review, we developed an approach to visualize a code change-set to review in the form of a graph. We implemented such an approach in a tool, called *ReviewVis*, which analyzes the code to be reviewed and extracts the static relations among classes and methods. *ReviewVis* focuses on the analysis of Java code and works together with GitLab.

Previous studies showed that highlighting the links between classes contained in a merge-request is a promising approach to support reviewers (Baum et al., 2017, 2019). For instance, Baum et al. focused on re-ordering files in a pull request based on their relations: *e.g.*, method calls or class hierarchies. However, ordering files may become unpractical when dealing with multiple relations at the same time. For instance, which relation should be prioritized over the others? Or how can these be made explicit to the reviewers? Furthermore, reviewers would still need to scroll and inspect the whole pull request to understand how the files relate to each other. For this reason, our tool aims to offer a summary of the content of a merge request under review. In this way, developers can immediately obtain an overall picture of all methods and classes that they have to review. This can increase their understanding of the code and guide them during the review.

3.1. Design requirements

To build our visualization tool for code review, we first consider existing literature to identify the requirements that the tool needs to fulfill. In the rest of this section, we discuss the requirements we identified and how these are implemented in our tool.

Support for review completeness and correctness: In a survey conducted by Tao et al. at Microsoft, the majority of software engineers reported how **information on change-set completeness and correctness are fundamental to achieve a good level of code understanding** (Tao et al., 2012). However, the developers also reported that acquiring information on the completeness of the code, as well as on its consistency and behavior, is often a challenging task.

[1] Gerrit: https://www.gerritcodereview.com.

Since previous studies showed that displaying links among classes and methods is beneficial to increase developers' understanding of the code (Bragdon et al., 2010; D'Ambros et al., 2009; Hanakawa, 2007), it seems reasonable to think that visualizing how different software entities are related to each other supports developers better understanding a change-set and assess its completeness and correctness.

For this reason, we decided to adopt a graph-based visualization approach, which constitutes the ideal approach to display relations among data (Munzner, 2014). Generally, graph-based techniques display entities as nodes, while the relations among them are represented as links in the graph. In our tool, we represented classes and methods as nodes and their relations (*e.g.*, method calls) as links.

Moreover, to increase the understanding support offered by the tool, we reduced the amount of possible confounding factors following a *force-directed placement* idiom (Munzner, 2014; Fruchterman and Reingold, 1991). Following this paradigm, each node is attracted to the center of the graph while being repulsed by other nodes. An auxiliary force keeps connected nodes close to each other. This approach minimizes the number of edge crossings and nodes overlaps (our implementation guarantees that nodes do not overlap) that can constitute distracting elements for the user.

Manage complex review data: A vast amount of links might exist among entities (classes and methods) in a review change-set. Therefore, displaying these relations in a clear way might become a challenging task. To help with this, interactions are fundamental: They allow to handle potential complexity in the data to visualize (Munzner, 2014). Furthermore, a single view can only show a static pre-determined version of the data. Interactions allow the user to customize how the data are shown: *e.g.*, highlighting different relations in the data. Given the importance of *interactivity* for visualization, we designed our tool accordingly: Developers can freely add or remove nodes in the graph and change the information displayed.

Display method calls and class relations: In a previous study, we investigated how to reorder review changes with the aim of supporting developers (Fregnan, 2018). In the context of that study, we conducted semi-structured interviews with 18 developers to understand which coupling relations are perceived as more important by developers to offer support in the context of code review. We asked developers to rank based on their importance for review eight relations: (1) method call, (2) declare-use, (3) object instantiation, (4) class inheritance, (5) parameter use, (6) field access, (7) same file, and (8) same file format. The interviewees ranked *method call* as the most importance relation, followed by *declare-use*, *object instantiation*, and *inheritance*.

Informed by these findings, we built our tool to use the following three coupling relations to construct the links in the graph: (1) class inheritance (including interfaces implementation), (2) object instantiations, and (3) method calls. We excluded the *declare-use* relation from our implementation as it is not well suited to work at class level and might introduce excessive complexity in the graph. To extract these relations, our tool analyzes the Abstract Syntax Tree (AST) of the code contained in the review change-set. The tool also analyzes links to classes and methods referenced in the merge request changes but not contained in the merge request itself. Developers can freely remove or add these nodes in the generated graph.

Integrate well with existing tools: The majority of developers do not like to use tools that interrupt their work flow: *e.g.*, by forcing them to switch from their current environment. In a previous study, Johnson et al. reported this to be one of the main reasons why developers choose not to use static analysis tools (Johnson et al., 2013).

Table 1
Coloring strategy used to highlight the change status of the nodes in *ReviewVis*'s graph.

Color	Node
	Added method or class
	Changed method or class
	Deleted method or class
	Generated method or class
	Unchanged method or class
	Non-java files

These findings highlight the importance of developing tools that are well-integrated with existing coding environments to not undermine their usability. Moreover, this paradigm was already successfully adopted by previous tools developed to support developers: *e.g.*, CARES (Guzzi et al., 2012).

For this reason, we decided to build our tool as an extension for the code review platform (GitLab) developers who took part in our in-company study already use for code review, instead of devising an independent tool.

A review change-set in GitLab is called *Merge Request* (the equivalent of a Pull Request on GitHub). A merge request involves a source branch (*i.e.*, the branch from where the merge request is created) and a target branch (*i.e.*, the branch where the merge request will be merged).

GitLab allows developers to review merge requests before they are merged into the project's codebase. Once a merge request is created, it is assigned to a reviewer who checks the code and adds line-level or design-level comments. When the review is complete, the merge request is assigned back to the author to allow them to address the reviewer's comments. Once the comments have been addressed, the process continues in an iterative fashion until all changes are deemed ready to be merged into the codebase.

3.2. ReviewVis walkthrough

Fig. 1 shows an instance of the graph created by *ReviewVis*. Rectangular nodes in the graph (part ① in Fig. 1) represent Java classes. Each class is marked with a label preceding the name of the class: *C* identifies a regular class, *A* marks an abstract class and, finally, *I* identifies an interface. A circle is added around all classes containing at least one method. Such circle (part ② in Fig. 1) allows the user to immediately identify which methods belong to a class. Methods are displayed in the graph as rectangles with rounded corners (*e.g.*, part ③). To allow developers to identify inner classes, *ReviewVis* shows the name of the class in which they are contained above the class node. Part ④ in Fig. 1 shows two inner classes *MainBox* and *TopBox* both contained inside *HelloWorldForm* class. Dotted lines represent method calls: *e.g.*, the method *getConfiguredLabel* of class *AddedStringField* calls method *get* of class *TEXTS* (part ⑤ in Fig. 1). Finally, non-java files (*e.g.*, part ⑥) are displayed with their full name, including the file extension.

Nodes are colored according to the strategy reported in Table 1. This strategy uses six different colors to show the status of a node: green (added nodes), orange (changed nodes), red (deleted nodes), gray (generated nodes: changed, added or deleted), white (unchanged nodes), and light blue (non-Java nodes).

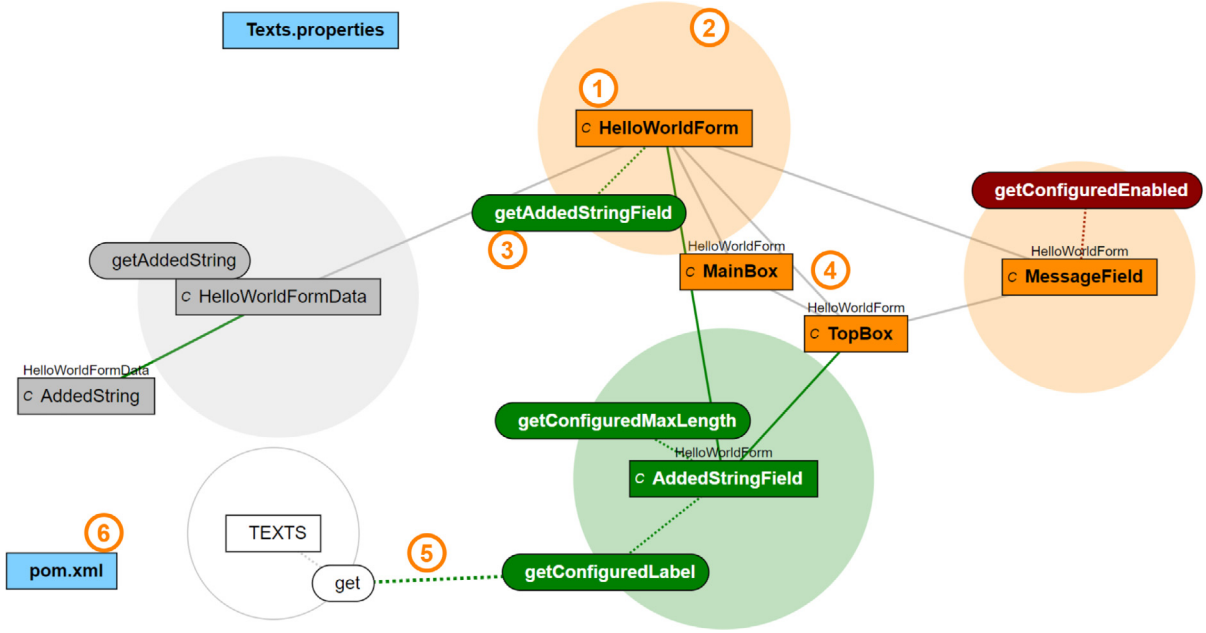


Fig. 1. Example graph generated for a GitLab merge request. The sample merge request was taken from the Eclipse Scout project (Eclipse Scout, 2020).

In the graph, the method nodes are shown in the proximity of the class implementing them (the distance between nodes does not convey any further information). Nodes representing inner classes (*e.g.*, the *TopBox* node in Fig. 1) are not contained inside the circle of the class node where they are implemented. This avoids increasing the complexity of the graph: An inner class node might also have a circle if the inner class contains a method modified in the considered merge request (*e.g.*, the *MessageField* node in Fig. 1, which represents an inner class of *HelloWorldForm*).

In the current design of *ReviewVis*, the graph does not report the direction of the method call relations. Even though the user can obtain this information from the code, the graph could also have also included it. However, in this stage of our research, we preferred not to add too many features to the current visualization to better understand how the basic ideas are perceived. Currently, reviewers can use the *graph-to-code* and *code-to-graph* navigation features of *ReviewVis* to easily move from the graph to the corresponding portion of code and vice versa. Nonetheless, future work should focus on assessing the benefits of integrating a directed method call relation in the graph of *ReviewVis*.

The graph created by our tool does not aim to substitute the inspection of the code of a merge request, rather to support it. To this aim, we designed *ReviewVis* to be used together with GitLab while reviewing the code. The graph provides reviewers with information on how the code entities (classes and methods) in the merge request to review are linked. As well as a visual summary of all the entities and how they participate in the change. This information aims to support reviewers in navigating (for instance, with the *graph-to-code* and *code-to-graph* features) and understanding the content of a review change-set.

3.3. ReviewVis *structure*

ReviewVis is formed by two main components: (1) a back-end component, CodeDiffParser, that extracts the relations in the review code and (2) a front-end component (CodeDiffVis) that constructs the graphs and allows the user to interact with it.

The decision to design *ReviewVis* as two clearly separated components was made to facilitate possible future extensions of *ReviewVis*. Currently, *ReviewVis* supports only the analysis of the Java files contained in a merge request. However, changing the parser implemented in the back-end component can allow the tool to be applied to code reviews containing files written in programming languages other than Java, without the need to reimplement the graph construction mechanism.

Back-end: CodeDiffParser. CodeDiffParser extracts the dependencies in the code under review and passes them in the form of a Json file to the front-end component. It uses the Eclipse AST parser to construct the Abstract Syntax Tree (AST) of the code in the merge request under analysis. It is able to analyze type declarations, method declarations, type instantiations, and method calls. More specifically, to construct the AST, the back-end relies on *ASTParser* and *HierarchicalASTVisitor* classes contained in the Eclipse parser.

CodeDiffParser performs this analysis for both branches (the source and target branch) of a merge request and compares the two resulting ASTs. The tool operates in the following way:

1. First, it analyzes the target branch (*i.e.*, the branch into which the merge request will be merged) and marks all nodes and links in the resulting AST as *deleted*.
2. Then, the source branch (*i.e.*, the branch from where the merge request will be merged) is analyzed and the resulting AST is created.
3. The target branch AST and the source branch AST are compared. Nodes present in both trees are marked as *unchanged* in the target branch tree. Nodes not present in the target branch AST but contained instead in the source branch AST are marked as *added*.
4. Finally, all parent nodes of added methods or inner classes are marked as *changed*.

The results are stored in a Json file using two lists for nodes and links, respectively. If the nodes contained in a branch of a merge request cannot be extracted, *ReviewVis* cannot generate the graph.

Front-end: CodeDiffVis. CodeDiffVis(CDV), implemented as a Google Chrome[2] browser extension, constitutes the front-end

[2] Google Chrome: https://www.google.com/intl/en_en/chrome/.

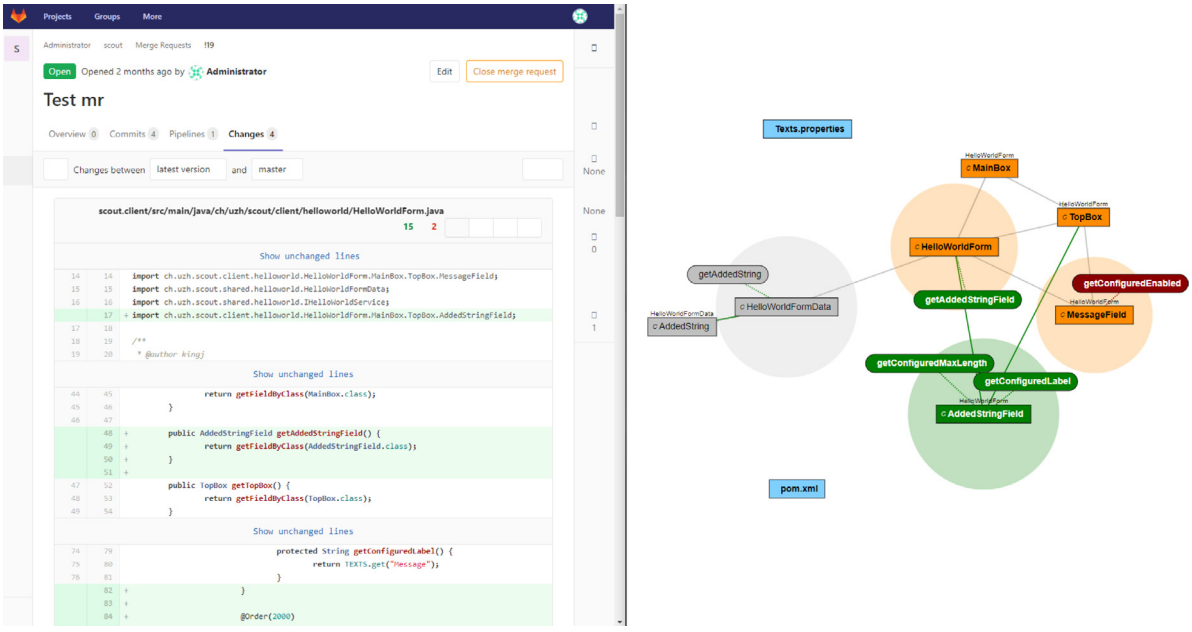


Fig. 2. View of the tool's front-end.

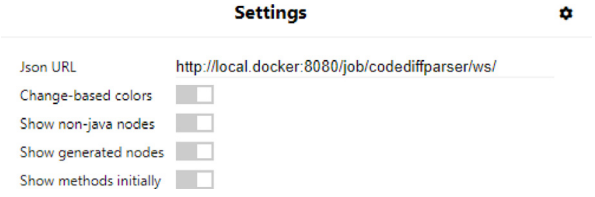


Fig. 3. Customization settings offered by *ReviewVis*.

part of our code review visualization tool. It is responsible for loading the Json file created by CodeDiffParser from either a local storage or a web service such as Jenkins.[3] We use *D3.js*[4] to create a force-directed graph which is – along with other JavaScript components – directly injected into the merge request page, using Google Chrome's content script programming paradigm. The graph is designed so that nodes cannot overlap. Fig. 2 shows a view of the tool graph next to the related merge request on GitLab.

The graph is displayed in a separate window. In this way, developers can place it according to their preferences (*e.g.*, in a separate screen). Furthermore, CodeDiffVis provides developers with a *setting* window (reported in Fig. 3) to customize the layout of the review changes graph. *ReviewVis* offers the following options:

Json URL: It allows the user to set the path to the Json file created by CodeDiffParser. The path can be either a local path or a web url.

Change-based colors: This option allows developers to switch between the two implemented color schemes. Nodes can be colored either by (1) *change status* or (2) *package*. The former coloring strategy is reported in Table 1. The latter color scheme uses the same color to represent nodes belonging to the same package. To this aim, we used a rainbow color map, adjusted to the number of packages to display.

Show non-java nodes: If enabled, this option displays in the graph both Java and non-Java files.

Show generated nodes: Automatically generated classes and methods might not require code review. However, they might still be relevant to understand the dependencies among entities in the merge request. For this reason, we allow users to enable or disable the visualization of automatically generated nodes in the graph.

Show methods initially: Hides/show method nodes in the graph. This option allows a reviewer to obtain a high-level overview of the relations among classes in the merge request under review, therefore, significantly reducing the size of the graph.

These interactions are fundamental to handle the complexity of the changes shown in the graph, as recommended by Munzner (2014). For instance, the user can hide some entities in the graph if this becomes too complex or if they are not interested in this information.

3.4. Interaction

ReviewVis allows the user to interact with the graph. In particular, a reviewer can freely alter the graph layout: *e.g.*, repositioning nodes, altering the size of the graph, or highlighting specific subgroups of nodes (subgraphs) in the graph. Furthermore, *ReviewVis* allows reviewers to navigate between the graph and the review code. There are two possible navigation interactions:

code-to-graph navigation: Hovering over the code in GitLab highlights the corresponding node in the graph. *ReviewVis* automatically centers the visualization on the highlighted node and adjust the zooming of the graph. Fig. 4 shows an example of *code-to-graph* navigation.

graph-to-code navigation: Clicking on a node in the graph allows the user to jump on the corresponding class or method in the GitLab merge request under review. Fig. 5 displays an example of the *graph-to-code* navigation.

[3] Jenkins: https://www.jenkins.io.
[4] D3.js: https://d3js.org.

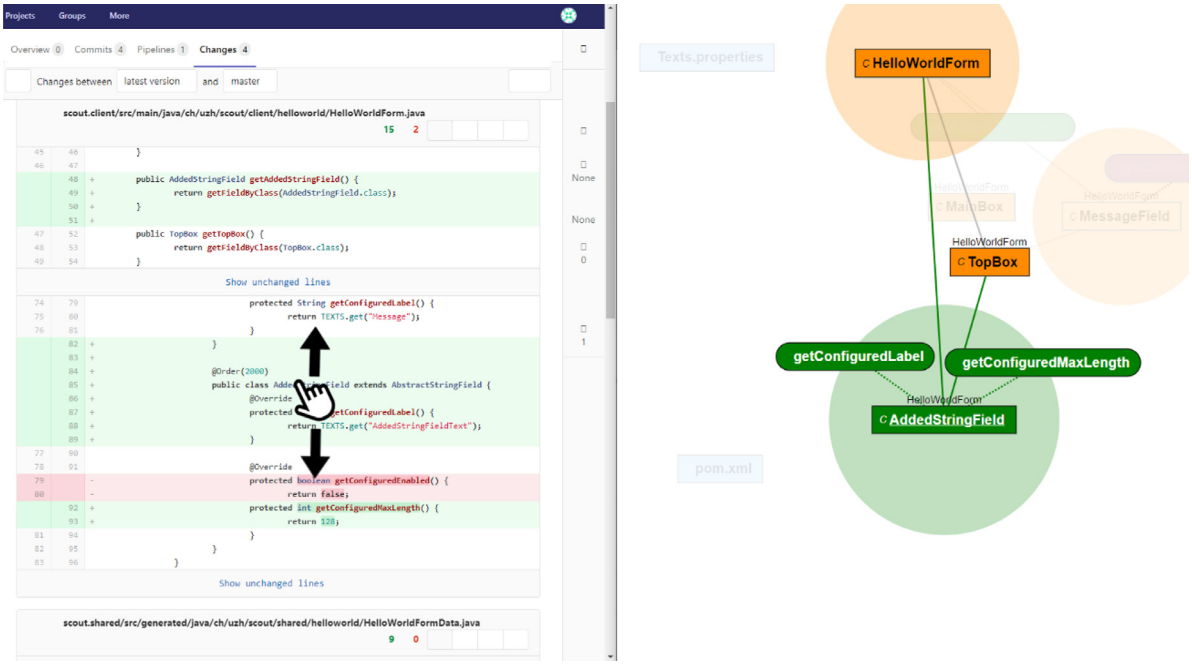


Fig. 4. Example of *code-to-graph* navigation. When a user hovers with the mouse on the class *AddedStringField*, the graph is centered on the corresponding node highlighting it and all nodes linked to it.

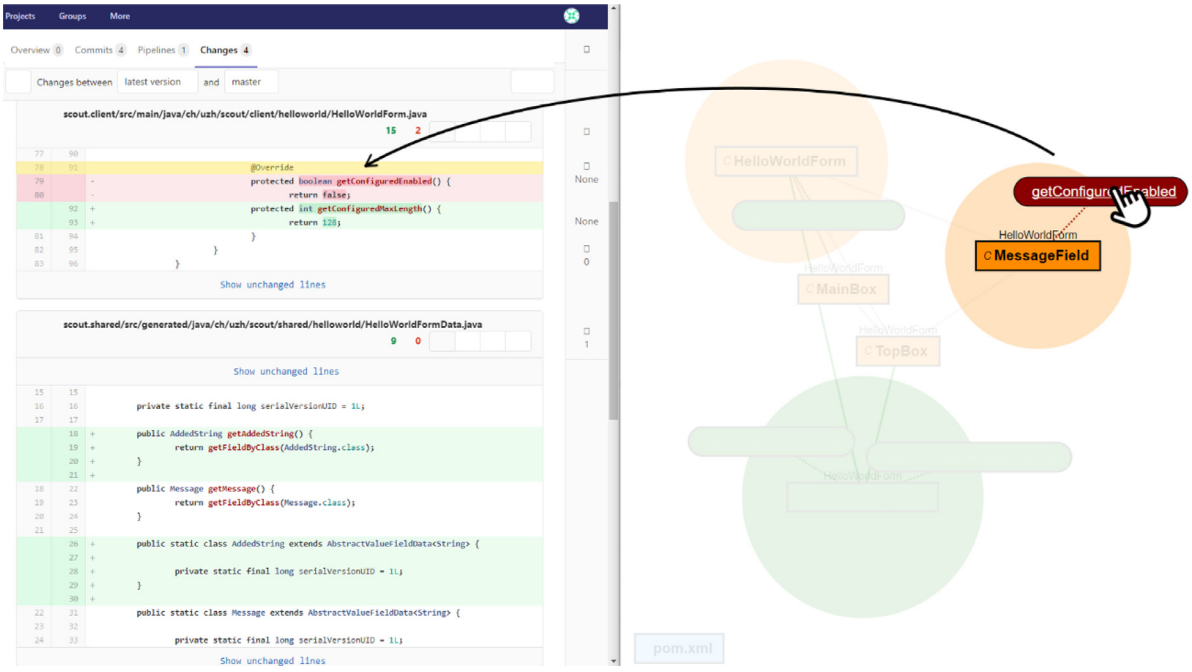


Fig. 5. Example of *graph-to-code* navigation. Clicking on a node in the graph shows the user the corresponding class or method definition in the associated GitLab merge request.

These two navigation features allow developers to move from a method/class in a merge request to the corresponding node in the graph or, on the contrary, from a node in the graph to the related portion of code. *ReviewVis* was developed as a support to the current code interface; therefore, we strived to enable developers to navigate from our tool to the source code diff and vice versa.

Previous work (Baum et al., 2017) showed how developers employ different strategies when reviewing code. For instance, reviewers may begin their review from the most important change parts. For this reason, we envision that developers can use the graph to identify the most prominent class in a merge request, then move to the code to begin the review while having an easy way to move back to the graph to check the interactions of that class with other entities in the merge request. It seems reasonable to think that this might be particularly useful to review large merge requests, where navigating the graph becomes more challenging for the reviewers.

Table 2 reports a list of all possible user interactions with the graph of *ReviewVis*. The interactions are categorized as (1) layout interactions, if they impact the layout of the graph, (2) navigation, if they allow the user to navigate from the graph

Table 2
Interactions in *ReviewVis*.

Name	Category	Description
Drag	Layout	Drag and drop nodes to change their position in the graph.
Zoom	Layout	Zoom in and out the graph to allow to focus on a specific group or nodes or obtain an overview of the relations in the code to review.
Node hovering	Layout	Hovering on a node automatically highlights all its nearest neighbors.
Hovering lock	Layout	Lock/Unlock the current highlighting of the graph.
Code-to-graph	Navigation	Hovering the mouse on a class or method in GitLab highlights the corresponding node in the graph. The graph automatically centers on the node and highlights it, together with all nodes linked to it.
Graph-to-code	Navigation	When the user clicks on a class or method in the graph, GitLab jumps to the corresponding declaration in the code to review. Furthermore, when a node is clicked, its appearance changes to allow a reviewer to keep track of the nodes already visualized/reviewed.
Expand	Custom	Show any connected referenced nodes.
Remove	Custom	Users can remove nodes from the graph.

to the code to review or vice-versa, and (3) custom when they allow customization of the graph to fit the user's preferences: *e.g.*, removing nodes.

4. Developers' perception

To evaluate *ReviewVis*, we first performed a two-step study to collect general feedback on its usability and benefits. This phase involved: (1) an in-company study asking professional software developers to use our tool and (2) an online survey with 37 software developers. Then, we focused on the developers' perception of *ReviewVis* to support their understanding of a review change-set. To this aim, we conducted a survey with 31 software developers. We spread our surveys broadly online to reach a heterogeneous sample of developers and avoid fixating, for instance, on specific review policies or review strategies. In the following sub-sections, we explain the data collection methods for both approaches, as well as the results.

4.1. In-company study — Methodology

We deployed our tool in a software development company that focuses on software development for business applications, having more than 320 employees in two countries. The company uses GitLab as tool to manage their repositories and employs Java as main programming language.

To recruit participants, we contacted developers through either (1) a request on the company internal board, (2) via email, or (3) directly via phone. Nine developers agreed to take part in our study.

Before conducting our investigation, we performed a pilot study with four professional software developers working in the same company. Our aim was to verify the goodness of the tool implementation, its correct deployment at the company, and the clarity of the online questionnaire. The participants of the pilot study did not join the in-company tool evaluation.

After improving the tool based on the collected feedback, we proceeded with the actual study. We asked participants (all professional Java developers) to use our tool to perform their code reviews. Since the number of merge requests that developers at the company have to review might significantly vary each week, we asked them to use our tool for two weeks. Participants received an installation guide and a brief tutorial on the functionalities of *ReviewVis* (available in our online replication package (Fregnan et al., 2020)). After the two weeks period, we asked them to complete an online questionnaire. A template of our questionnaire is available in our replication package (Fregnan et al., 2020).

The structure of the online questionnaire given to participants was the following:

Welcome page: We introduced participants to the aim of the online questionnaire as well as the goal of our research. Before participants were allowed to answer the questions, we asked them to agree to the treatment of their data according to the European General Data Protection Regulation (GDPR). Moreover, we assigned a randomly generated ID to each participant to allow them to ask the removal of their data from our dataset at a later time.

General open questions: At the beginning of our questionnaire, we asked developers about their experience with *ReviewVis*. The questions were posed as open-text questions. We asked participants (1) what they appreciated the most (and the least) of our visualization tool and (2) for which review changes *ReviewVis* was most (and the least) effective. In this step, we asked a total of four open-test questions to the participants. Our goal was to collect general feedback on the tool before asking participants to focus on specific aspects of the tool.

Usability questions: To assess the usability of *ReviewVis*, participants were asked questions using an adapted version of the System Usability Scale (SUS) defined by Brooke (Brooke, 1996). Our aim was to evaluate the usability of *ReviewVis* to ensure that the collected feedback was not biased by defects in the tool's usability.

Features questions: After collecting general feedback on *ReviewVis*, we asked developers to evaluate the information and the interaction with the tool's graph using a five-point Likert scale. The statements evaluated by developers are reported in Figs. 7 and 8. Moreover, we included a *I do not know* answer to cover cases where a participant could not evaluate a specific feature. The goal of these questions was to evaluate developers' perception of the graph and the information offered by it (*e.g.*, were they easy to understand?) after they used it in their code reviews.

Applicability and benefits of *ReviewVis*: In this part of the questionnaire, we asked participants to evaluate the applicability of *ReviewVis* and its possible benefits. Figs. 9, 10, and 8 report the statements participants were asked to evaluate. With these questions, we aimed to collect feedback on the possible benefits of *ReviewVis* and its applicability.

Demographics: At the end of the questionnaire, we collected participants' demographics as well as their experience as programmers and reviewers.

Final remarks and conclusion: In the last page of the survey, participants were asked to enter any final remark they had about our tool or the questionnaire itself. This question was not mandatory. Also, we asked participants' consensus to share their answers in an anonymized publicly available research dataset.

Table 3
In-company study participants' demographics (N = 9). ND = 'Not Disclosed'.

Participant	Gender	Coding exp. (years)	Java exp. (years)	Review exp. (years)
P1	Male	2	2	2
P2	Female	≤1	≤1	≤1
P3	Male	≥11	≥11	≥11
P4	Male	6–10	6–10	6–10
P5	Male	6–10	6–10	6–10
P6	Male	3–5	3–5	2
P7	ND	6–10	6–10	2
P8	Male	≥11	≥11	3–5
P9	Male	3–5	3–5	3-5

At the end of each group of questions, we included an optional open-text question to collect possible further feedback from the participants.

To analyze the answers that developers gave to the open questions regarding their experience with *ReviewVis*, we used open card sorting (Spencer, 2009). This technique allowed us to extract recurring opinions on what worked best with *ReviewVis* or, instead, which characteristics of our tool were not appreciated by developers. One of the authors first performed the card sorting, leading to the creation of a set of labels. Then, a second author independently performed the card-sorting process using the existing set of labels. The two authors achieved an inter-rater agreement of 87.5%. In case of a disagreement, the two authors involved in this process started a discussion until an agreement was reached. This approach helped us to strengthen the confidence in the results of this process, reducing the bias caused by unclear or mislabeled comments.

4.2. In-company study — Results

Nine professional developers took part in our study. Table 3 reports gender, general coding experience, coding experience with Java, and code review experience of the participants. Furthermore, participants spent on average ≈28 h per week coding (std.: 8.3) and ≈5 h per week reviewing code (std.: 3.28).

Participants' answers to the usability of *ReviewVis* are reported in Fig. 6. We computed the System Usability Scale (SUS) score following the guidelines provided by Lewis (2018). Overall, *ReviewVis* achieved an average score of 71.11 (std. 11.97). This result confirms the goodness of our implementation. A poor implementation might have negatively influenced developers' feedback on the visualization approach proposed by *ReviewVis*. However, to further improve the usability of *ReviewVis*, two participants mentioned how the integration with Gitlab should be improved. For instance, P9 stated: "If [ReviewVis] were a bit easier to use … I could see myself using it on a daily basis. The Gitlab MR interface is horribly inefficient by default".

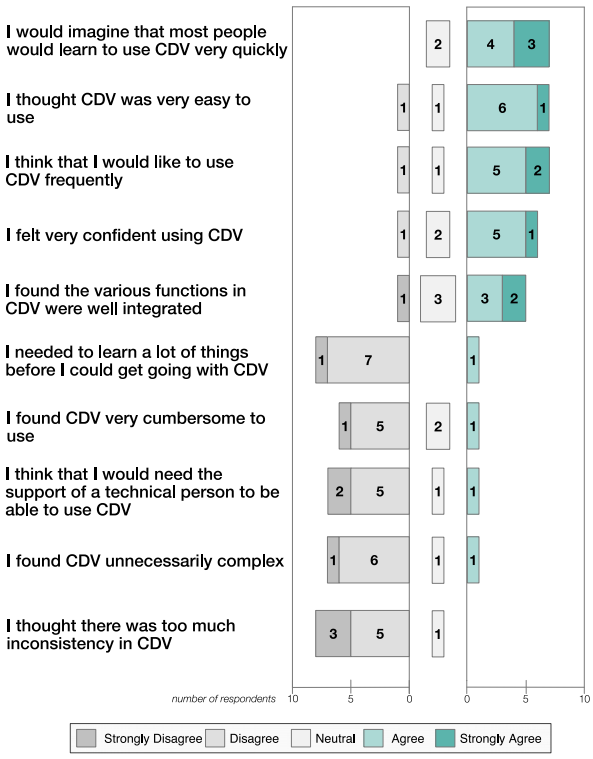


Fig. 6. Participants' answers to the System Usability Scale questions used to assess *ReviewVis*. CDV indicates CodeDiffVis, the front-end component of our tool that displays the graph.

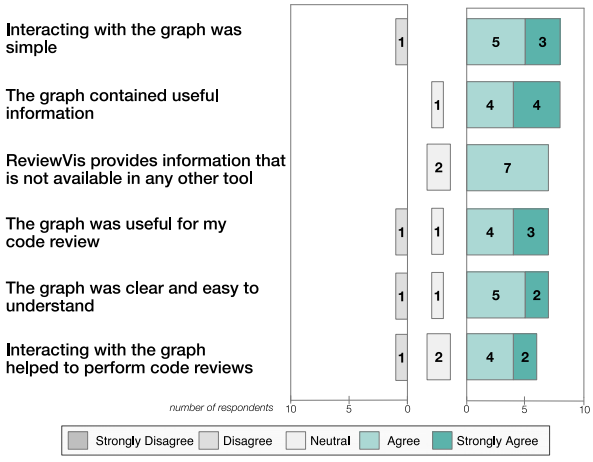


Fig. 7. Participants' evaluation of the graph created by our tool. Neutral includes both *Neutral* and *I do not know* answers.

4.2.1. Graph representation

Fig. 7 reports participants' answers concerning the graph created by *ReviewVis*. Overall, the majority of participants evaluated the graph positively. Seven developers declared to not have had issues in understanding the graph (*the graph was clear and easy to understand*) and eight reported that interacting with the graph was simple.

Moreover, eight participants acknowledged the usefulness of the information offered by the generated graph (*the graph contained useful information*), while one developer gave a neutral answer concerning this statement. To further confirm the benefits of the graph to perform code review, we asked developers to evaluate if the information offered by the graph and interacting with it helped them while reviewing code. Seven developers reported that the graph gave them support to perform code review (*the graph was useful for my code review*), while six recognized the benefits of interacting with the graph during code review (*interacting with the graph helped me to perform code review*).

4.2.2. Goodness of information displayed by the graph

Fig. 8 reports participants' answers for each of the questions contained in this section of the questionnaire. Overall, developers positively assessed the amount of information displayed by *ReviewVis* (*the graph displays sufficient information*): Six developers agreed with our statement, while three were neutral about it. Then, we asked participants to focus on specific features of the graph to understand how developers perceive them. Overall, participants acknowledged the usefulness of the features of the

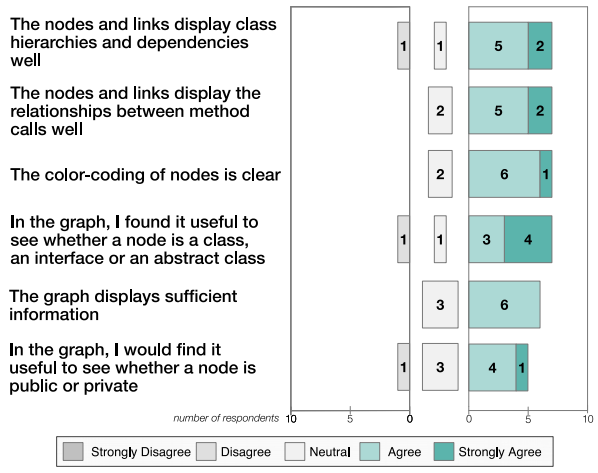


Fig. 8. Participants' evaluation of the information displayed by the graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

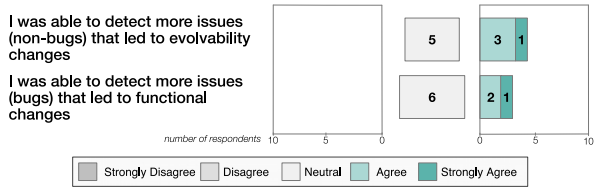


Fig. 9. Participants' evaluation of the kind of defects (evolvability or functional) that *ReviewVis* supports in finding. Neutral includes both *Neutral* and *I do not know* answers.

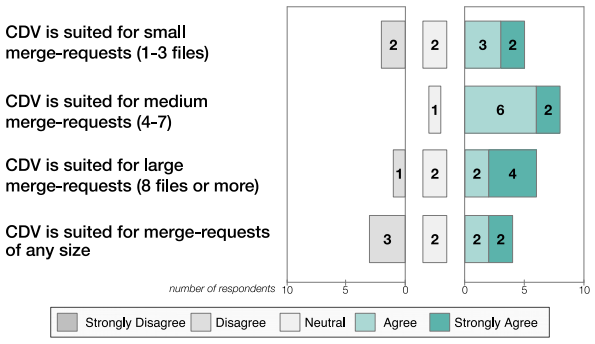


Fig. 10. Participants' evaluation of the applicability of *ReviewVis* in terms of size of the merge requests. CDV indicates CodeDiffVis, the front-end component of our tool that displays the graph.

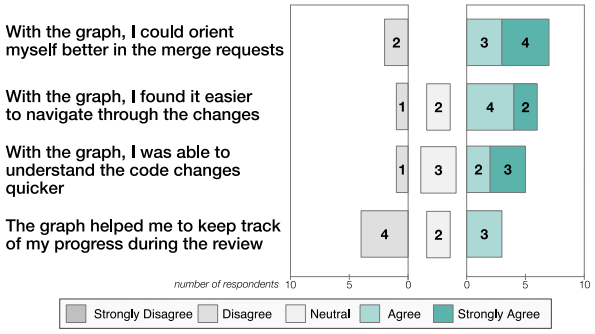


Fig. 11. Participants' evaluation of the benefits of *ReviewVis*. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

graph. In particular, the graph was perceived as effective in displaying information on the method calls between different nodes. Moreover, developers positively evaluated the color scheme used to represent nodes with different statuses (*e.g.*, added or removed nodes).

4.2.3. Benefits and applicability

In the following section of the questionnaire, we asked participants to evaluate the benefits and the applicability of *ReviewVis*. First, we asked them if using our tool to perform code review they think they have been able to identify more maintainability defects or more functional defects compared to their usual code reviews. We reported their answers in Fig. 9. Concerning maintainability issues, four developers reported that our tool helped them to find more of this kind of defects. Three developers gave neutral answers to this statement, while two were unsure. When asked if our tool helped them in identifying more functional defects in the code, three developers replied positively, while five gave neutral answers and one was unsure. P2 further commented on this aspect: "I feel it's easier to handle evolvability changes with CodeDiffVis than functional behavior. I think for a better view on the functional changes you would need some sort of representation of runtime behavior". Another participant (P5) reported: "I would not say that CodeDiffVis leads to more detection per se but it leads to a better understanding of the connections between classes, methods and such, therefore leading to a deeper understanding of the code. This, of course, could lead to more detection".

Fig. 10 reports participants' answers when asked to evaluate the applicability of *ReviewVis*. The majority of participants agreed that *ReviewVis* achieves the best results when applied to medium-size merge requests (merge requests containing between four to seven files). Developers' perception on the goodness of our tool to review large merge requests (with eight or more files) was still positive, while only slightly positive for small merge requests (with three or less files). It seems reasonable to think that when dealing with small merge requests, developers can already easily notice links among the classes and methods they have to review, therefore reducing the effectiveness of the support offered by *ReviewVis*. This interpretation was confirmed by participants' comments: *e.g.*, P3 reported: "It seems to me that merge requests with four changes don't need extra tooling. It's when I get a merge request with 30 changes that I wish for better tooling", while P9 remarked: "For 1–3 files, I'm not sure if a graph is actually needed or useful. The larger the Merge-request, the more useful I think".

Finally, we asked participants to evaluate four possible benefits of using *ReviewVis* to perform code review, which we report in Fig. 11 together with developers' answers. Developers positively evaluated the support of our tool in helping understand ('With the graph, I was able to understand the code changes quicker') and navigate the code ('With the graph, I could orient myself better in the merge requests' and 'with the graph, I found it easier to navigate through the changes'). Four participants disagreed when asked if the graph also helped to keep track of their progress during the review.

4.2.4. Overall pros and cons

When asked (through open-text questions) about what they liked/disliked the most about *ReviewVis*, participants reported having appreciated the possibility to see an overview of the code to review and its dependencies (6 mentions) as well as the possibility to navigate from the graph to the code and vice versa using the *code-to-graph* and graph-to-code features (2 mentions). For example, P9 reported: "Seeing which classes/methods were

added/removed/used is great to "map out" the scope of the merge request". However, developers suggested to take action to improve the layout of the graph (4 mentions): *e.g.*, P9 answered: "The graph is not "stable" and moves around. While the interactivity is great for small MRs (*merge requests*), in daily use with large MRs a graph that is more 1d than 2d and less interactive would probably be more useful". Participants reported different aspects of the review in which they found *ReviewVis* had been beneficial: *e.g.*, to understand the context of a merge request (2 mentions). On the contrary, participants reported that the specific focus on Java files (2 mentions) and on the changed parts of the code (2 mentions) limited the tool's applicability in their code reviews.

4.2.5. Follow-up interviews

To collect further feedback on our tool, we performed follow-up semi-structured interviews with two developers who took part in the in-company study. The interviews were performed as a video call over the Internet and lasted approximately 30 min each. During the interviews, we asked the participants about the features they particularly appreciated of *ReviewVis* and those they disliked, with a particular focus on the interactions offered by our tool.

Concerning the tool interactions, one participant (PI1) reported that they were not so intuitive and he would have preferred to be able to understand them without having to use the tutorial provided during the in-company study. For instance, PI1 mentioned that the color scheme of the graph was not clear to him at first. Furthermore, the participant expressed that *ReviewVis* generated too many nodes since the tool creates nodes for both classes and methods modified in a merge request. On the contrary, the second participant in the interview (PI2) reported that the amount of nodes was reasonable, but noticed how the layout of the nodes might not have been optimal because the nodes of the graph were shown too close to each other. Moreover, PI2 reported to have particularly appreciated the *graph-to-code* feature of *ReviewVis*.

To further understand the reason behind developers' evaluation of the interactive features of *ReviewVis*, we performed a second survey with Java developers. We report our findings in Section 4.5.

4.3. General evaluation of ReviewVis — Methodology

To complement our findings from the in-company study, we performed an online survey. We organized the survey following a similar structure to the one employed for the in-company questionnaire. We spread the survey through various social media platforms (*e.g.*, Twitter and Reddit). In the survey, we presented short videos highlighting the main features of the tool to the participants as a way to see it in action without having to install it on their machines. Another possible option would have been to allow participants to download the tool and try it locally. However, we deemed this possibility unpractical since it would have significantly increased the time required to complete the survey. Although using short videos (no longer than 20 s) allowed us to reduce the completion time of the survey, this still remained fairly complex. The presence of videos and images of the tool, necessary to give participants an overview of its main functionalities, might have reduced the number of developers willing to participate in our survey. Our survey and the asked questions are available in our replication package (Fregnan et al., 2020).

The structure of the survey is the following:

Welcome page: Participants are introduced to the topic and the goal of the survey: the evaluation of *ReviewVis*. Moreover, each participant is assigned a uniquely generated ID to allow them to ask for the removal of their answers from the dataset at a later time.

Questions on visualization for code review: In the introductory part of the survey, our aim was to collect feedback on developers' perception of visualization tools and techniques to support code review. First, participants were provided with the definition of the concepts of *visualization* and *visualization tools*. Then, developers were asked to evaluate if code review would benefit from the use of visualization tools and explain why such tools would (would not) be helpful. Finally, we asked developers if they know or use any visualization tools for code review. In case of a positive answer, participants were asked to explicitly state the tools they know/use.

Questions on *ReviewVis*: In this section, we asked participants to evaluate *ReviewVis* and its specific features: Each feature was shown in a video before each specific question together with captions and a brief explanation of the feature. Participants were given the possibility to re-watch the whole video or a specific part of it. Furthermore, before answering the questions, participants were provided with an overview of the tool. First, we asked participants about the usefulness of the graph and its features. Then, we asked participants to evaluate the layout, code-to-graph, graph-to-code, and customization features of *ReviewVis*. The statements that participants had to evaluate are reported in Figs. 13, 14, and 15. The goal of these questions was to collect developers' feedback on the main features of *ReviewVis*. We wanted to (1) ensure that the information and features of the graph are easy to understand and (2) collect developers' perception on their usefulness for code review. Finally, we asked developers to evaluate possible benefits of using our visualization tool: *e.g.*, having at their disposal information not available in any other tool or being supported in understanding the code changes to review. Figs. 15 and 16 report the statements developers were asked to evaluate. The main goal of these questions was to collect feedback on the potential benefits of *ReviewVis*.

Demographics questions: As in the questionnaire used for the in-company study (Section 4.1), we collected participants' demographics as well as information on their experience in programming and code review.

Final remarks and conclusion: Finally, we asked participants to state any final remarks on our tool or the survey. Furthermore, we asked participants if they were willing to allow us to publish their answers in an anonymized publicly available research dataset.

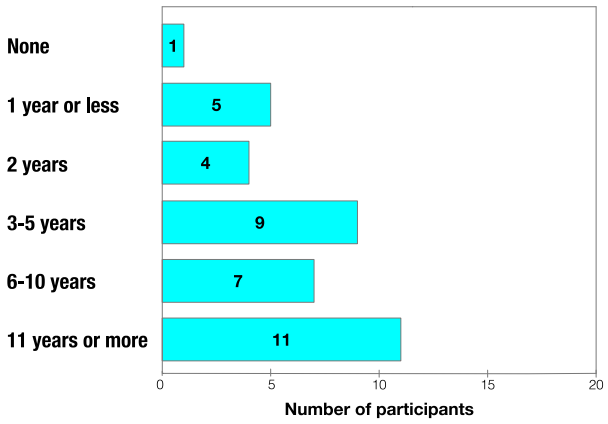
4.4. General evaluation of ReviewVis — Results

Our survey received 37 valid answers. Fig. 12 reports participants' experience in developing software in a professional setting, developing in Java, and performing code review.

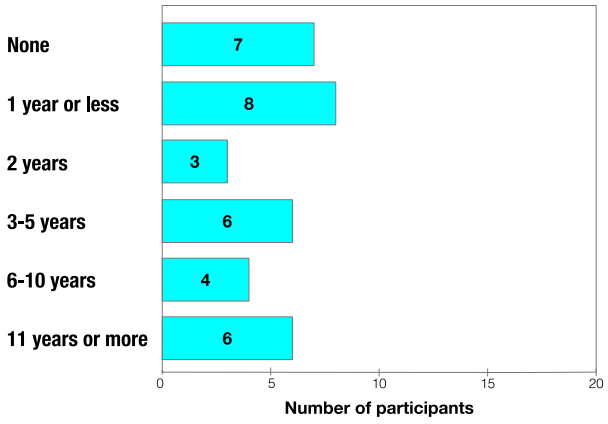
At the beginning of the survey, we asked developers if they know or use any visualization tool for code review. With these questions, our goal was to understand the adoption of visualization tools for code review: For instance, are they something developers normally use while performing code review? In the survey, 18 developers reported knowing other visualization tools (other 18 respondents replied they did not know any visualization tool and one did not answer), but only 12 declared to use such tools to perform their reviews (24 developers replied they did not use any visualization tool to perform code review and one did not answer). When asked to specify which tool they use, participants mentioned either tools meant to visualize and analyze source code but without a specific focus on code review (*e.g.*, *HAPAO* (Bergel and Peña, 2014)), or tools such as Phabricator[5] or GitLab,[6] which cannot be consider proper visualization tools. This

[5] Phabricator: https://www.phacility.com/phabricator/.

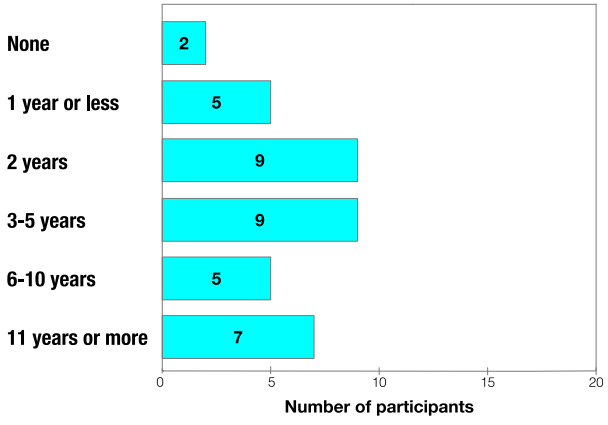
[6] Gitlab: https://about.gitlab.com.



(a) Participants' experience with software development (N = 37).



(b) Participants' experience with Java (N = 34).



(c) Participants' experience with code review (N = 37).

Fig. 12. Survey participants demographics.

result shows the lack of visualization tools aimed specifically at code reviews. Moreover, it supports our claim that code review visualization is still a vastly unexplored research area.

Fig. 13 reports participants' answers when asked to evaluate the graph created by *ReviewVis*. The majority of participants positively assessed the goodness of the color-coding scheme used in the graph and the importance of the label displaying whether a node is a class, an interface, or an abstract class. These results are in line with our findings from the in-company study (Section 4.2).

When asked if the graph displays sufficient information and is easy to understand, the majority of participants replied positively.

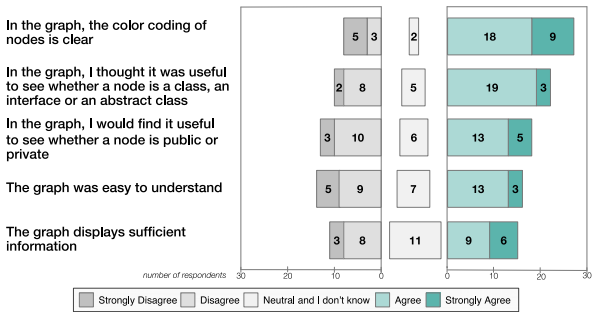


Fig. 13. Participants' evaluation of the information displayed by the graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

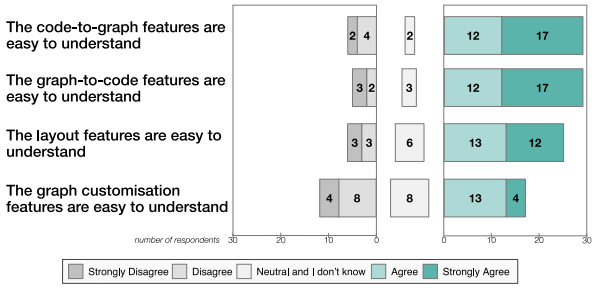


Fig. 14. Participants' evaluation of the comprehensibility of the visualization features of *ReviewVis*.

However, 31% and 38% of the participants, respectively, disagreed with this statement. Furthermore, a non-negligible amount of developers (26% and 19%) chose a neutral answer. These results are far from the one collected during the in-company study (Section 4.2), where 78% of the participants judged the graph clear and easy to understand and 89% reported that the graph contained useful information to perform code review. Such a difference might have been caused by different settings of the two questionnaires: Participants in the in-company study had the possibility to use our tool in their normal working routine, while participants in the online survey were only briefed on the tool and its functionalities but could not interact with it.

Similarly, when participants were asked to evaluate the usefulness of displaying in the graph whether a node is public or private, the majority of developers (49%) agreed on the usefulness of such a functionality, although a significant part of them negatively evaluated it (35%) or took a neutral stance (16%).

In the online survey, participants were asked to evaluate four key features of *ReviewVis*: (1) layout features (drag, zoom, node hovering, and hovering lock; an extensive explanation of these features is reported in Table 2), (2) graph-to-code, (3) code-to-graph, and (4) graph customization features (add connected nodes, and removed nodes from the graph). Participants' answers concerning the comprehensibility of these features are reported in Fig. 14. Fig. 15 reports developers' answers concerning the usefulness of the main features of *ReviewVis* (and the tool itself) for code review. All the four presented sets of features were positively assessed by the participants in the online survey.

Concerning the *code-to-graph* and *graph-to-code* features, the majority of developers (78%) confirmed that they were easy to understand, while 70% and 72% of the online survey participants thought that these features might help them to perform code review. The layout features were defined as easy to understand by 69% of the participants, but only 50% of the participants reported that they might be useful for code review with a further

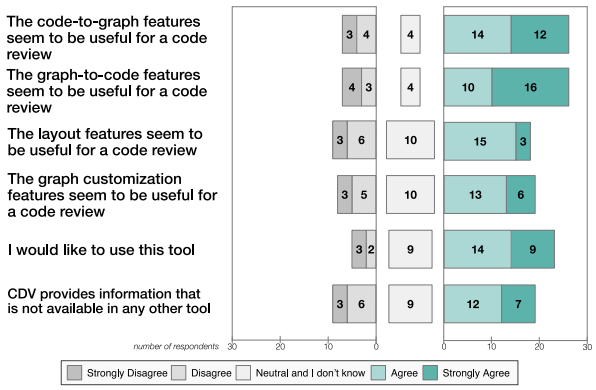


Fig. 15. Participants' evaluation of the usefulness of *ReviewVis* and its features. CDV indicates CodeDiffVis, the front-end component of our tool that displays the graph.

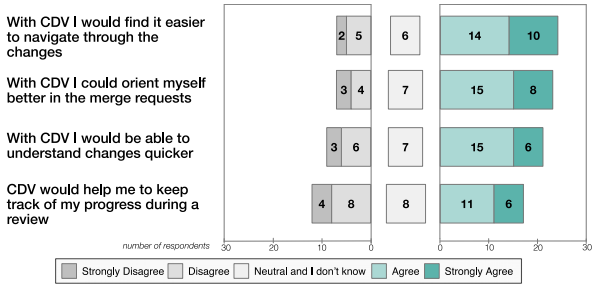


Fig. 16. Participants' evaluation of the comprehensibility of review changes with *ReviewVis*. CDV indicates CodeDiffVis, the front-end component of our tool that displays the graph.

25% of developers who took a neutral stance. Finally, the graph customization features were reported as the most problematic to understand (47% of the participants agreed on the statement that they were easy to understand), but, nonetheless, they were judged as useful for code review by 58% of the participants.

Developers were also asked to evaluate the possible benefits of using *ReviewVis* to perform code review and to assess the novelty of the information shown. Fig. 16 shows participants' answers on the benefits of *ReviewVis* to increase their comprehension of code changes during review. The most positive aspect of the tool appears to be its ability to support developers in navigating through the changes and in the merge-request: These statements were judged positively by the 65% and the 64% of the participants, respectively. Moreover, 60% of the developers reported that *ReviewVis* helped them in understanding the changes faster than without the information provided by our tool. When asked if they would use our tool in practice, 62% of the developers responded positively (*Agree* or *Strongly agree*). Finally, participants' answers were divided when asked if *ReviewVis* helped them in keeping track of their progress during code review. Although most of the developers (46%) answered positively to this statement, a significant percentage of them were unsure (22%) or answered negatively (32%). These results confirm our findings from the in-company study. Further work seems to be necessary to make the proposed tool useful for developers to keep track of their progress during the review.

4.5. ReviewVis support for understanding — Methodology

Our findings from the in-company study and survey evaluated *ReviewVis* in terms of usability and its potential benefits for developers during code review. Given the positive results, we investigate developers' perceptions on how they think *ReviewVis* can support their understanding of a review change-set. To this aim, we designed a second online survey. Similar to the online survey presented in Section 4.3, we presented short videos highlighting the main features of the tool to the participants. This survey is available in our replication package (Fregnan et al., 2020).

We structured the survey as follows:

Welcome page: We introduce the goal and topic of our investigation to the participants. To mitigate the risk of moderator acceptance bias, we state that our aim was to evaluate a visualization tool created by researchers rather than disclosing our identity as the creators of the tool. Moreover, all participants are requested to approve our data handling policy and are assigned a unique identifier that they can use to request the removal of their answers from our dataset at a later time.

Presentation of ReviewVis: In this section, we present *ReviewVis* and ask the participants to evaluate whether the graph is easy to understand and the color coding is clear. This was motivated by our follow-up interviews (presented in Section 4.2.5), which revealed that these might be problematic aspects of *ReviewVis*.

ReviewVis support for understanding: In this section, we ask participants to evaluate whether they think the features and interactions offered by *ReviewVis* are beneficial to increase their understanding of a review change-set. We present the features using short videos and a brief textual explanation. To test the clarity of the explanations and encourage participants to pay attention, after each video we include a question that asks the participants information on the feature.

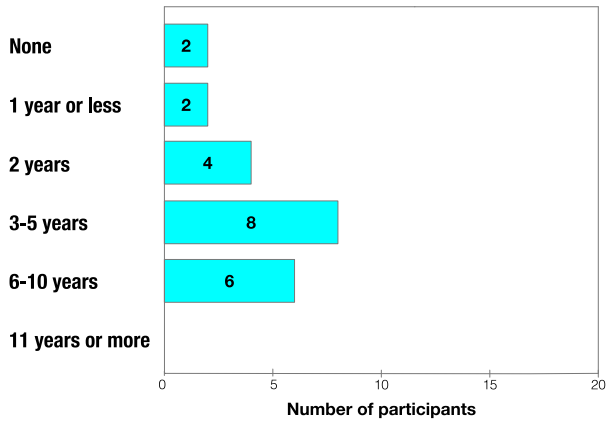
Benefits of ReviewVis and TAM: Subsequently, we ask respondents to evaluate other possible benefits of our tool. Moreover, we assess the participants' perception of *ReviewVis* applying the Technological Acceptance Model (TAM) (Davis, 1989). This model assesses the usefulness and ease of use of a newly devised technology.

Demographics questions: As in the in-company study questionnaire (Section 4.1) and previous survey (Section 4.3), we collect participants' demographics and information on their experience in programming and code review.

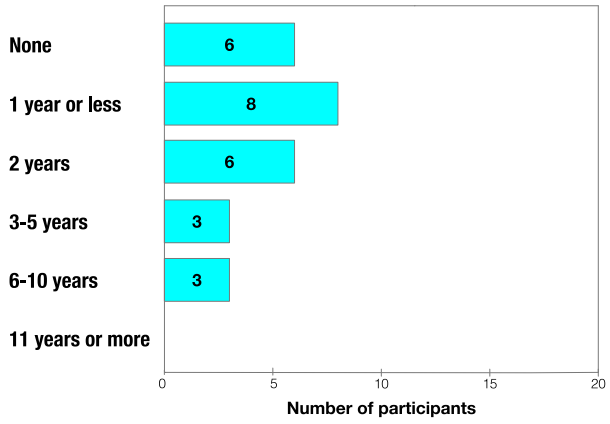
Final remarks and conclusion: To conclude the survey, we ask the participants' consensus to publish their answers in an anonymized publicly available research dataset. Moreover, we ask them about any final remarks on our tool or the survey.

4.6. ReviewVis support for understanding — Results

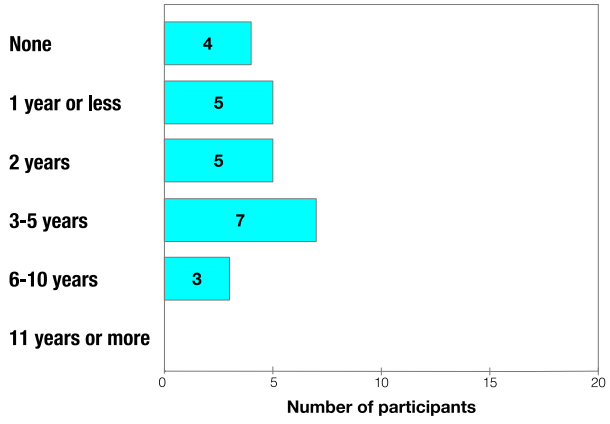
We spread the survey through social media platforms (*e.g.*, Twitter and Reddit) as well as the personal network of the authors. We collected answers for a period of time of four weeks. Our survey received 31 valid answers. Fig. 17 reports the experience with software development, Java, and code review of the participants, while Fig. 18 illustrates the frequency at which participants currently perform both programming and code review. Among the participants, 23 identified themselves as males, three as female, and five chose either to not disclose this information or to self-define. Moreover, two participants reported having already taken part in our previous survey on *ReviewVis*. We decided not to exclude the answers of these respondents as this survey has a different focus compared to the first one (reported in Section 4.3). Moreover, these participants are more familiar with the tool,



(a) Participants' experience with software development in a professional setting (N=22).

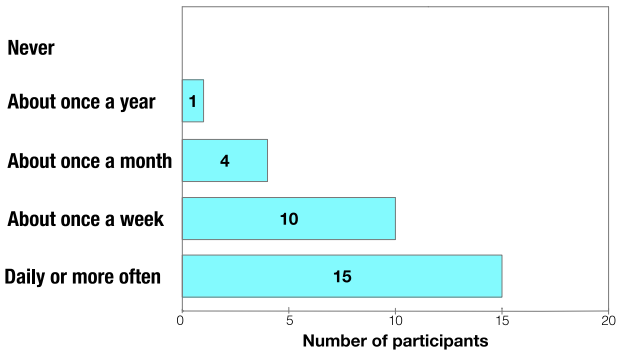


(b) Participants' experience with Java in a professional setting (N=26).

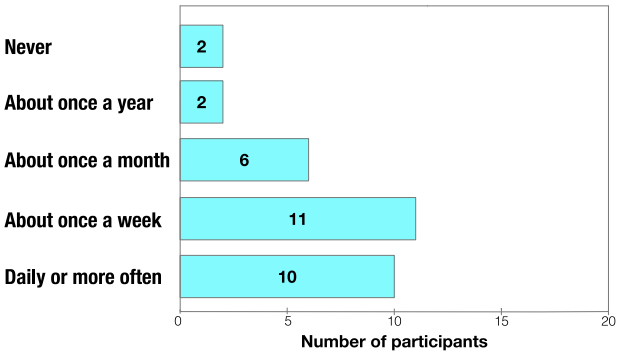


(c) Participants' experience with code review in a professional setting (N=24).

Fig. 17. Demographics of the participants in the understanding survey.



(a) Frequency at which participants currently do programming (N=30).



(b) Frequency at which participants currently perform code reviews (N=31).

Fig. 18. Demographics of the participants in the understanding survey.

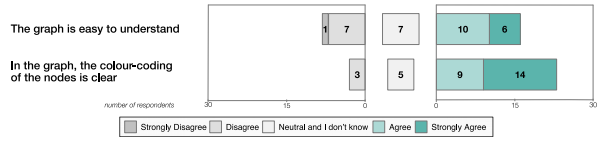


Fig. 19. Participants' answers to the general questions concerning the graph created by *ReviewVis*. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

which, in turn, might reduce potential bias in their answers caused by a poor understanding of the tool's features.

Evaluation of the graph. At the beginning of the survey, after providing participants with an explanation of our tool, we asked them to evaluate if (1) the graph was easy to understand and (2) the color-coding used to represent the nodes was clear. Our goal was to collect information on participants' perception of the understandability of our tool as well as to conduct an initial evaluation of our graph as a whole without focusing on specific features. The results obtained are reported in Fig. 19. The majority of the respondents (≈74%) confirmed the goodness of the color coding used in the graph. However, although the majority (≈52%) of the participants evaluated positively the understandability of the graph, they did not reach a strong consensus. This result may have been influenced by the fact that we could provide participants with neither a detailed explanation nor a practical test of *ReviewVis* in our survey.

Features' impact to support understanding. In the next section of the survey, we asked participants to evaluate (1) the potential effect of each feature of *ReviewVis* on their understanding of a review change-set and (2) the role that the offered interactions play in supporting their understanding. Fig. 20 reports respondents' evaluation of each feature of *ReviewVis*.

The majority of the participants positively assessed the *code-to-graph* and *graph-to-code* features to support their understanding. When asked to motivate the reason behind their answers, participants reported: "I can easily maintain a direct "location" between the source code and the graph" and "It seems like a handy "link" to jump directly to the code, without having to scroll through all the changes to find what I'm looking for".

The *layout* feature was also evaluated positively by the majority of the participants, although a larger amount of them chose

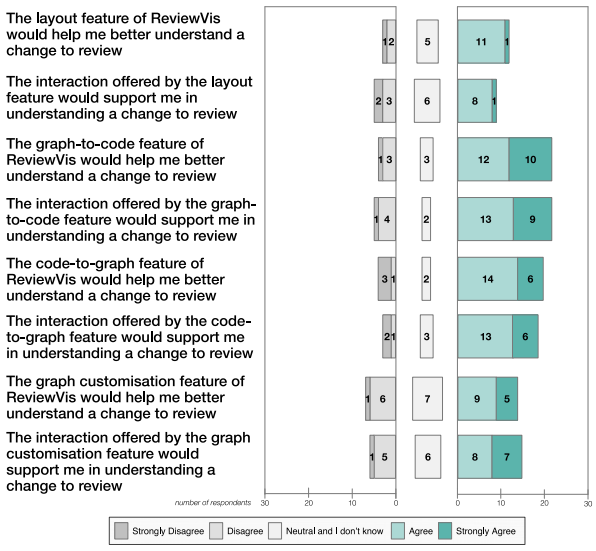


Fig. 20. Participants' evaluation of each feature and interaction of *ReviewVis* to support their understanding of a review change-set.

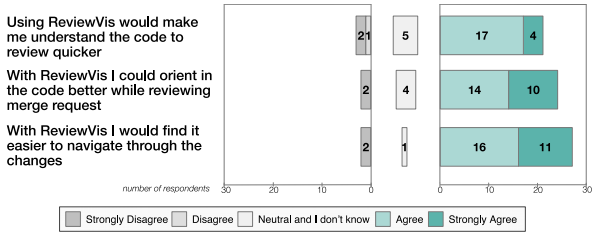


Fig. 21. Participants' evaluation of the potential benefits of *ReviewVis*.

a neutral answer compared to the previous features. Finally, respondents gave a lukewarm evaluation of the potential effects of the *customization* features of *ReviewVis* on their understanding of a review change-set. A participant who positively evaluated this feature reported: "It allows to systematically go through the changes in more complex scenarios and dig deeper or remove unnecessary parts of the graphs". On the other hand, participants who negatively assessed the impact of the *customization* feature reported that this feature constitutes only another factor to which they have to pay attention while performing their reviews, thus, increasing their cognitive load. For instance, a participant stated: "One more complex thing to manage; might remove something accidentally".

Benefits of ReviewVis. Fig. 21 shows participants' evaluation of the potential benefits of *ReviewVis* to understand and navigate a merge request. Overall, the vast majority of the participants (80%) acknowledged how our tool would support them while reviewing code. In particular, 90% of the respondents positively valued the help offered by *ReviewVis* in navigating through the review changes of a merge request. This is in line with the findings of the more general survey presented in Section 4.3. This result supports the positive value of a graph-based visualization to support reviewers.

Technological Acceptance Model. In the last section of our survey, we employed the Technological Acceptance Model (TAM) to evaluate (1) the perceived usefulness of *ReviewVis* and (2) its perceived ease of use. Fig. 22 reports the answers of the participants concerning the usefulness of our visualization tool. Most participants found *ReviewVis* useful. All items presented less

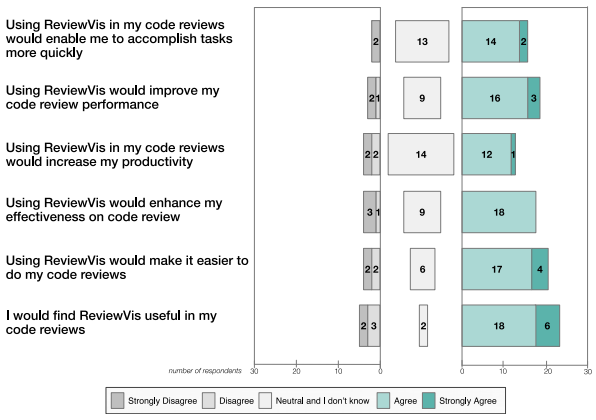


Fig. 22. Participants' evaluation of the perceived usefulness of *ReviewVis*.

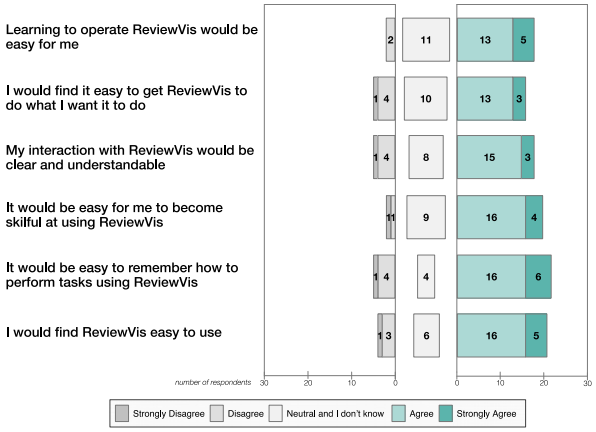


Fig. 23. Participants' evaluation of the ease of use of *ReviewVis*.

than 20% negative answers and were, overall, positively evaluated by the majority of the respondents. Nonetheless, items such as 'Using ReviewVis in my code reviews would enable me to accomplish tasks more quickly' or 'Using ReviewVis in my code reviews would increase my productivity' registered a number of neutral answers close to 50%. This high number may be caused by the difficulties that participants faced in evaluating the potential impact of *ReviewVis* on their productivity without being able to try the tool first-hand.

Fig. 23 shows participants' answers concerning the perceived ease of use of our tool. Overall, the majority of developers agreed that *ReviewVis* would be easy to use. Nonetheless, certain items registered a high number of neutral answers (*e.g.*, the 'Learning to operate *ReviewVis* would be easy for me' item). As for the perceived usefulness, it is possible that the respondents struggled to evaluate the approach without using it, therefore took a neutral stance.

Overall, the results suggest that developers perceive a graph-based visualization approach as having the potential to support developers during code review and to increase their understanding of a change-set.

5. Discussion

In this section, we discuss how our results lead to recommendations for practitioners and designers, as well as implications for future research.

Understanding and review effectiveness. Our qualitative study gave an initial indication that a tool implementing a graph visualization in the form similar to what is proposed by *ReviewVis* could

be welcomed by industry developers. Their perception of the tool was positive, praising its usability and benefits. Developers reported that the graph gave them support to perform code review, helping them in better understanding the code change and in navigating the code.

Studies targeted at evaluating improvements on code review performance normally focus on (Baum et al., 2019; Spadini et al., 2019; Thongtanunam et al., 2014) *review effectiveness* (*i.e.*, number of defects found) and *review efficiency* (*i.e.*, number of defects found in a unit of time) (Biffl, 2000). Our hypothesis is that a better understanding of the code under review (as potentially achievable with our visualization) can lead to increased review performance. However, to verify this hypothesis and the effect of our tool, new studies, such as controlled experiments, need to be designed and carried out. In our study, we focused on assessing the usability and the perceived benefits of a graph-based visualization approach to support reviewers. In particular, our focus was on supporting developers' understanding of a review change-set. The positive results of our evaluation strengthen our confidence in the positive potential of our tool (*e.g.*, assuring that its benefits are not undermined by a poor usability), therefore paving the way towards the design of further evaluation of its benefits. We envision, for instance, a controlled experiment to evaluate *ReviewVis*'s impact on developers' review effectiveness and efficiency.

Support for multiple languages. Currently, *ReviewVis* only supports Java. However, participants in our study reported how this implementation choice might limit the usefulness of the tool. To solve this problem, we envision the use of the Language Server Protocol (LSP).[7] An LSP implementation exists for almost any language, and it is possible to have more language servers running at the same time (this feature makes it possible to have *ReviewVis* working even when the project uses more than one programming language). Finally, since the company where we deployed the tool used GitLab, we developed the tool on top of it. We expect the porting of our tool to other platforms to be without major roadblocks because *ReviewVis* relies on the same data that most code review tools offer through their APIs.

The impact of Merge Requests' size. One recurring topic that emerged from developers' feedback concerned the applicability of *ReviewVis*. According to the participants of our study, *ReviewVis* was perceived as especially useful when applied to Merge Requests of medium-size (4–7 files), while it was perceived as less useful on small-size (1–3 files) or big-size (8 + files) ones. As previously reported, the reason is that developers perceive that a small graph, *i.e.*, with only two nodes and one edge, does not carry useful enough information. In fact, the developers stated that they could easily understand the links between the classes by simply reading the code. On the other hand, when more than eight files are involved in the code change, developers reported perceiving the graph as becoming too big, with too many nodes and edges that make it hard and time-consuming to get information out of it. A large graph might also affect reviewers with information overload, negatively impacting their review performance.

To better understand the potential impact of *ReviewVis*, we analyzed a total of 138,452 pull requests from 208 Java-based open-source projects on GitHub. We selected these projects among the most popular java-based projects on GitHub, with a star-count above 1000. Based on previous work (Blincoe et al., 2016; Borges et al., 2016), we used stars as a representation of the popularity and health of a project. We chose a large set of projects to reduce potential bias caused by project-specific policies or characteristics (the complete list of projects is available in our replication package (Fregnan et al., 2020)).

Our goal was to understand how many pull requests belong to each of the three defined categories: small, medium, and large PRs. Our analysis revealed that 16.72% of the pull requests belong to the *medium* category (having between 4 and 7 files), while the vast majority (67.7%) can be defined as *small* pull requests. We argue that the amount of *medium* pull requests, for which developers perceived our tool would be particularly helpful, is non-negligible and make the problem of supporting developers during the review of these pull request (by definition, more complex than the small ones) particularly important. Further studies should be devised to investigate the support offered by our tool specifically for *medium sized* PRs.

Our participants perceived that our tool offered only limited support when reviewing small PRs as they are easy to understand even without external help. When it comes to larger PRs (eight or more files), further improvements of the visualization are needed to better display this kind of changes: Currently, the graph might become too complex for large PRs, hindering its benefits to support developers' understanding. To display large merge requests, different visualization paradigms might be better suited: *e.g.*, a city-based approach, as the one proposed in *ViDI* (Tymchuk et al., 2015) or *CodeCity* (Wettel and Lanza, 2008). A city-based paradigm offers to developers a more compact view of software classes compared to a graph-based visualization approach. This makes it suitable to visualize large Merge Requests, giving reviewers an immediate overview of all the classes to review, while this information might be hard to immediately read from a graph. However, the use of a city-based model would not allow to clearly show relations among code entities: Including such relations might overwhelm the user with too complex information (Jeffery, 2019). Further studies are needed to find a suitable visualization paradigm to clearly display to reviewers the relations between code entities in a large merge request.

Visualization approaches developed for software evolution also constitute a precious source of inspiration to tackle this issue. For instance, the visualization paradigm of *Evolution Radar* (D'Ambros et al., 2009) might achieve promising results as a means to support code review. *Evolution Radar* offers an effective way to visualize the strength of the links between different classes. Including this approach in a review visualization could support the inspection of large merge requests: Once a class is selected, the tool makes immediately clear which other classes are strongly linked to the selected one and, therefore, should be inspected next.

Another promising source of inspiration is the work of Benomar et al. (2013). The authors used heat maps to display information about software evolution. In the context of *ReviewVis*, a heat map might be applied to the graph nodes to convey information about the importance of each class. This would guide developers while reviewing a merge request, telling them where to start the inspection of the code: A node colored in red will most likely be the ideal starting point of the review.

Moreover, *ReviewVis* currently does not suggest to reviewers in which order to review the files contained in a merge request. However, such a feature might reduce the cognitive load for reviewers, especially when dealing with large merge requests. For this reason, in future investigations, we envision combining our tool with approaches to identify the most salient classes in a review change-set, as the one proposed by Huang et al. (2018).

Visualization and guidance. Participants in our study acknowledged the benefits of our visualization approach to navigate and

[7] https://langserver.org/.

understand code changes. However, they took a neutral stance on the use of *ReviewVis* to keep track of their progress during the review. A reviewer's progress during code review can be measured in two ways: (1) keeping track of the reviewed classes or (2) keeping track of checked topics. To include the former in our visualization approach, we might employ a different node color-coding scheme to distinguish between already checked classes and classes yet to be inspected by the reviewers. The color of a node could be changed automatically by the tool once a class has been visited (similar to read/unread emails in a traditional email client) or manually by the reviewer. To keep track of checked topics instead, we envision combining our graph-based visualization with review checklists. Code review checklists allow developers to mark the items they already checked during the review (McConnell, 2004; Rong et al., 2012; Gonçalves et al., 2020). This raises the following question: How can we integrate such a feature in a visualization paradigm? This is no trivial task as it might significantly increase the complexity of the visualization and, consequently, undermine its usability. Investigating how checklists and visualization can be combined effectively appears to be a valuable area for future research.

6. Limitations

Implementation bias. The choice of focusing on GitLab to develop our tool might have introduced bias in the results. Although we cannot exclude that combining *ReviewVis* with another review tool might lead to different results, GitLab does not present any significant difference with other popular code review tools (*e.g.*, GitHub or Gerrit) in the way in which the review is conducted.

Company review policies. During the *in-company* study, developers used *ReviewVis* to perform the code review tasks assigned to them as part of their normal responsibilities. Although this allowed us to collect valuable feedback on the use of our tool in a real-case scenario, we had no control over the number and size of the reviews the participants performed. To mitigate this issue, we asked participants to use our tool for a period of two weeks to increase the variety in terms of size and complexity of the reviews they performed.

Participants sample bias. To collect feedback on the proposed solution, we deployed our tool in a software development company. Although participants had different programming and code review experience, we cannot exclude that common policies in place at the company might have introduced bias in our results. To mitigate this threat, we extended our feedback collection on the tool through an online survey.

Only one female developer participated in the in-company study. Previous studies (Beckwith et al., 2006; Burnett et al., 2011) reported how participants' gender might influence their approach to problem-solving. This might have negatively influenced the generalizability of our results.

Despite participants overall positively assessed the benefits of *ReviewVis*, we still registered a significant disagreement between them. Therefore, although our investigation achieved positive results indicating *ReviewVis* as a promising tool to support reviewers, further investigations are needed to fully understand the benefits offered by our tool.

Survey lack of interaction. In the online survey, participants could not try the tool in a real-case scenario, but its features were explained to them through videos. This might have introduced bias in participants' understanding of the tool's functionalities. To mitigate this threat, each feature of the tool was explained in a separate video. We kept the video duration short, so participants were required to focus only for a limited amount of time.

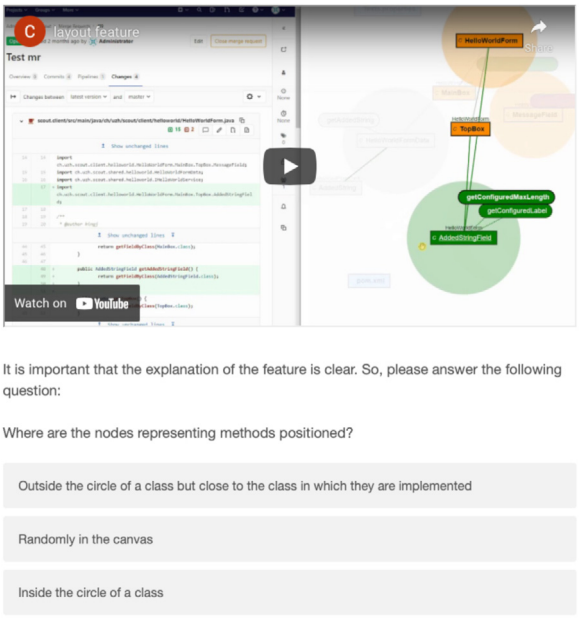


Fig. 24. Example of question asked to assess participants' understanding of *ReviewVis* features. The video above (that participants are asked to watch) contains the explanation of the feature necessary to correctly answer the understanding question. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We explicitly asked participants to watch the videos carefully. To further support participants' understanding, each video contained captions explaining the specific functionality. Moreover, we used a neutral language to describe the features to avoid leading participants towards selecting positive votes. Despite these measures, our results might still suffer from bias caused by participants difficulties in understanding the features or not paying attention to the videos. To further mitigate this threat, in our follow-up survey (presented in Section 4.5) we included a question after each video to evaluate participants' understanding of the provided explanation and we excluded for a specific feature the answers of the participants who did not reply correctly or skipped the question. Fig. 24 shows an example of the questions used to evaluate participants' understanding.

Comparison with a baseline. Participants in the in-company study used *ReviewVis* in combination with Gitlab to perform their code review as opposed to using Gitlab alone. Overall, developers did not report significant difficulties when using the tool and assigned to *ReviewVis* a System Usability score of 71.11. These results give an initial indication of the good usability of our tool compared to a baseline (Gitlab). However, in our study we did not *explicitly* investigate how *ReviewVis* compares to Gitlab and whether it poses additional information load on the reviewers. To foster developers adoption of such review visualization tools, it is paramount to avoid the presence of steep learning curves and high cognitive effort. Further studies should be devised to assess, for instance, the cognitive overload our approach might pose on reviewers. Moreover, the feedback provided by developers (*e.g.*, to strengthen the tool integration into Gitlab) should be addressed to further improve the usability of *ReviewVis*.

Adoption for middle-sized merge requests. Developers might not be willing to adopt and learn how to use this tool if they perceive it as useful only for middle-sized merge requests, as this constitutes only a limited part of the merge requests they have to review. However, the positive feedback collected in the

in-company study suggests that developers would be willing to adopt the tool, despite it being perceived as less useful for small or large change-sets, as long as it is well integrated into their existing workflow. Moreover, the good System Usability Scale (SUS) score achieved by our tool indicates how ReviewVis would not present a steep learning curve for developers. These results are corroborated by the results of our understanding survey (Section 4.6), where participants acknowledged the easiness of use of *ReviewVis* through a Technological Acceptance Model (TAM).

Currently, *ReviewVis* needs to be activated by the user (through a button in the browser) to display the graph. This lets developers decide when to use the tool, preventing it from overloading developers with information when code review support is not needed. However, future studies can be designed and carried out to determine whether it is possible to automatically show/hide the view, based on the characteristics of the review change-set (*e.g.*, its complexity). This would increase the ease of use of our tool and strengthen its integration into code review tools.

7. Conclusion

In this paper, we presented a graph-based visualization approach to support developers during code review. Classes and methods are represented as nodes in the graph, while the links represent the structural coupling relations among them. We implemented the visualization (*ReviewVis*) as a Google Chrome extension that displays the classes and methods in a GitLab merge request.

We collected general feedback on *ReviewVis* through an in-company study with nine professional software developers and an online survey, collecting 37 valid answers. Participants in the *in-company* study used our tool to perform code review as part of their normal working routine. They positively assessed the usefulness of the information reported in the graph as well as the benefits of interacting with the tool while reviewing code: *ReviewVis* helped developers in understanding and navigating through the changes to review. These findings were corroborated by the feedback collected from the online survey participants.

Given the positive results of this first evaluation, we conducted a second survey with 31 participants with a specific focus on the potential benefits of *ReviewVis* to support the understanding of the content of a merge-request.

Overall, our investigation highlighted how a graph-based visualization technique is perceived as beneficial to support developers in reviewing change-sets, in particular, as a way to increase their understanding of the changes to review. Based on our findings, further studies should focus on quantifying the benefits offered by visualization for code review: *e.g.*, in terms of review effectiveness and efficiency. New visualization techniques could be employed to address the current limitations of *ReviewVis* (*e.g.*, when dealing with large merge requests).

Our results indicated how *ReviewVis* is particularly suited to support reviewers in medium-sized PRs, while the support it offers might be negligible for small PRs. For large PRs, our tool's benefits are currently limited by the tool's implementation. Further investigations are needed to devise approaches to better display large changes, avoiding excessive complexity.

Moreover, further studies should be conducted to verify that *ReviewVis* does not pose significant information load on developers, which might hinder its adoption in the future and reduce its benefits on supporting reviewers.

Overall, our investigation and tool constitute only an initial step towards creating a production-ready visualization approach to support code review. To the best of our knowledge, only a very limited amount of studies focused on devising visualization tools to support reviewers. It is our hope that the insights we collected can pave the road towards devising further approaches to better support developers' understanding of a change-set at code review time.

CRediT authorship contribution statement

Enrico Fregnan: Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing. **Josua Fröhlich:** Conceptualization, Software, Investigation, Writing – original draft. **Davide Spadini:** Validation, Formal analysis, Writing – original draft. **Alberto Bacchelli:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Availability of data and material

All data and materials are available in our replication package at the following link: https://doi.org/10.5281/zenodo.7047993

Acknowledgment

E. Fregnan and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111506.

References

Ackerman, A.F., Buchwald, L.S., Lewski, F.H., 1989. Software inspections: an effective verification process. IEEE Softw. 6 (3), 31–36.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 712–721, URL: http://dl.acm.org/citation.cfm?id=2486788.2486882.

Barnett, M., Bird, C., Brunet, J., Lahiri, S.K., 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 134–144.

Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016a. Factors influencing code review processes in industry. In: Proceedings of the 2016 24th Acm Sigsoft International Symposium on Foundations of Software Engineering. pp. 85–96.

Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016b. A faceted classification scheme for change-based industrial code review processes. In: 2016 IEEE International Conference on Software Quality, Reliability and Security. QRS, pp. 74–85. http://dx.doi.org/10.1109/QRS.2016.19.

Baum, T., Schneider, K., 2016. On the need for a new generation of code review tools. In: International Conference on Product-Focused Software Process Improvement. Springer, pp. 301–308.

Baum, T., Schneider, K., Bacchelli, A., 2017. On the optimal order of reading source code changes for review. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 329–340. http://dx.doi.org/10.1109/ICSME.2017.28.

Baum, T., Schneider, K., Bacchelli, A., 2019. Associating working memory capacity and code change ordering with code review performance. Empir. Softw. Eng. 24 (4), 1762–1798. http://dx.doi.org/10.1007/s10664-018-9676-8.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., Cook, C., 2006. Tinkering and gender in end-user programmers' debugging. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 231–240.

Bedu, L., Tinh, O., Petrillo, F., 2019. A tertiary systematic literature review on Software Visualization. In: 2019 Working Conference on Software Visualization. VISSOFT, IEEE, pp. 33–44.

Benomar, O., Sahraoui, H., Poulin, P., 2013. Visualizing software dynamicities with heat maps. In: 2013 First IEEE Working Conference on Software Visualization. VISSOFT, IEEE, pp. 1–10.

Bergel, A., Peña, V., 2014. Increasing test coverage with hapao. Sci. Comput. Program. http://dx.doi.org/10.1016/j.scico.2012.04.006.

Biffl, S., 2000. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In: Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000. IEEE, pp. 136–145.

Blincoe, K., Sheoran, J., Goggins, S., Petakovic, E., Damian, D., 2016. Understanding the popular users: Following, affiliation influence and leadership on GitHub. Inf. Softw. Technol. 70, 30–39.

Borges, H., Hora, A., Valente, M.T., 2016. Understanding the factors that impact the popularity of GitHub repositories. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 334–344.

Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., Laviola, J.J., 2010. Code bubbles: A working set-based interface for code understanding and maintenance. In: Conference on Human Factors in Computing Systems - Proceedings. http://dx.doi.org/10.1145/1753326.1753706.

Brooke, J., 1996. SUS: a "quick and dirty'usability. In: Usability Evaluation in Industry. CRC Press, p. 189.

Burnett, M.M., Beckwith, L., Wiedenbeck, S., Fleming, S.D., Cao, J., Park, T.H., Grigoreanu, V., Rector, K., 2011. Gender pluralism in problem-solving software. Interact. Comput. 23 (5), 450–460.

D'Ambros, M., Lanza, M., Lungu, M., 2009. Visualizing co-change information with the evolution radar. IEEE Trans. Softw. Eng. 35 (5), 720–735.

D'Ambros, M., Lanza, M., Robbes, R., 2010. Commit 2.0. In: Proceedings of the 1st Workshop on Web 2.0 for Software Engineering. pp. 14–19.

Davis, F.D., 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q. 319–340.

Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S., 2015. Untangling fine-grained code changes. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 341–350.

Diehl, S., 2007. Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer Science & Business Media.

2020. Eclipse Scout project. https://github.com/eclipse/scout.rt.

Eick, S.G., Steffen, J.L., Sumner, E.E., 1992. Seesoft—A tool for visualizing line oriented software statistics. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/32.177365.

Fagan, M.E., 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. 15 (3), 182–211. http://dx.doi.org/10.1147/sj.153.0182.

Francel, M.A., Rugaber, S., 2001. The value of slicing while debugging. Sci. Comput. Program. 40 (2–3), 151–169.

Fregnan, E., 2018. Automatic ordering of code changes for review.

Fregnan, E., Fröhlich, J., Spadini, D., Bacchelli, A., 2020. Replication package. https://doi.org/10.5281/zenodo.7047993.

Fruchterman, T.M., Reingold, E.M., 1991. Graph drawing by force-directed placement. Softw. - Pract. Exp. 21 (11), 1129–1164.

Gasparini, L., Fregnan, E., Braz, L., Baum, T., Bacchelli, A., 2021. ChangeViz: Enhancing the GitHub pull request interface with method call information. In: 2021 IEEE Working Conference on Software Visualization. VISSOFT, IEEE.

Gómez, V.U., Ducasse, S., D'Hondt, T., 2015. Visually characterizing source code changes. Sci. Comput. Program. 98, 376–393.

Gonçalves, P.W., Fregnan, E., Baum, T., Schneider, K., Bacchelli, A., 2020. Do explicit review strategies improve code review performance? In: Proceedings of the 17th International Conference on Mining Software Repositories.

Gousios, G., Pinzger, M., Deursen, A.v., 2014. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. pp. 345–355.

Guzzi, A., Begel, A., Miller, J.K., Nareddy, K., 2012. Facilitating enterprise software developer communication with CARES. In: 2012 28th IEEE International Conference on Software Maintenance. ICSM, IEEE, pp. 527–536.

Hanakawa, N., 2007. Visualization for software evolution based on logical coupling and module coupling. In: 14th Asia-Pacific Software Engineering Conference (APSEC'07). IEEE, pp. 214–221.

Huang, Y., Jia, N., Chen, X., Hong, K., Zheng, Z., 2018. Salient-class location: help developers understand code change in code review. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 770–774.

Jeffery, C.L., 2019. The city metaphor in software visualization.

Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R., 2013. Why don't software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 672–681.

Khaloo, P., Maghoumi, M., Taranta, E., Bettner, D., Laviola, J., 2017. Code park: A new 3D code visualization tool. In: Proceedings - 2017 IEEE Working Conference on Software Visualization, VISSOFT 2017. http://dx.doi.org/10.1109/VISSOFT.2017.10, arXiv:1708.02174.

LaToza, T.D., Arab, M., Loksa, D., Ko, A.J., 2020. Explicit programming strategies. Empir. Softw. Eng. 1–34.

Lewis, J.R., 2018. The system usability scale: past, present, and future. Int. J. Human–Comput. Interact. 34 (7), 577–590.

MacLeod, L., Greiler, M., Storey, M.-A., Bird, C., Czerwonka, J., 2017. Code reviewing in the trenches: Challenges and best practices. IEEE Softw. 35 (4), 34–42.

Mattila, A.-L., Ihantola, P., Kilamo, T., Luoto, A., Nurminen, M., Väätäjä, H., 2016. Software visualization today: Systematic literature review. In: Proceedings of the 20th International Academic Mindtrek Conference. pp. 262–271.

McConnell, S., 2004. Code Complete. Pearson Education.

Munzner, T., 2014. Visualization Analysis and Design. CRC Press.

Oosterwaal, S., Deursen, A.v., Coelho, R., Sawant, A.A., Bacchelli, A., 2016. Visualizing code and coverage changes for code review. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1038–1041.

Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact. 2 (CSCW), 135:1–135:27. http://dx.doi.org/10.1145/3274404, URL: http://doi.acm.org/10.1145/3274404.

Rigby, P.C., Bird, C., 2013. Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2013, ACM, New York, NY, USA, pp. 202–212. http://dx.doi.org/10.1145/2491411.2491444, URL: http://doi.acm.org/10.1145/2491411.2491444.

Rong, G., Li, J., Xie, M., Zheng, T., 2012. The effect of checklist in code review for inexperienced students: An empirical study. In: 2012 IEEE 25th Conference on Software Engineering Education and Training. IEEE, pp. 120–124.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., Bacchelli, A., 2018. Modern code review: A case study at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. In: ICSE-SEIP '18, ACM, New York, NY, USA, pp. 181–190. http://dx.doi.org/10.1145/3183519.3183525, URL: http://doi.acm.org/10.1145/3183519.3183525.

Shull, F., Seaman, C., 2008. Inspecting the history of inspections: An example of evidence-based technology diffusion. IEEE Softw. 25 (1), 88–90.

Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., Bacchelli, A., 2019. Test-driven code review: An empirical study. In: Proceedings of the 41st International Conference on Software Engineering. ICSE '19, IEEE Press, Piscataway, NJ, USA, pp. 1061–1072. http://dx.doi.org/10.1109/ICSE.2019.00110.

Spencer, D., 2009. Card Sorting: Designing Usable Categories. Rosenfeld Media.

Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S., 2012. How do software engineers understand code changes? An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

Tao, Y., Kim, S., 2015. Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 180–190.

Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H., 2014. Improving code review effectiveness through reviewer recommendations. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM, pp. 119–122.

Tymchuk, Y., Mocci, A., Lanza, M., 2015. Code review: Veni, ViDI, Vici. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 151–160.

Wang, M., Lin, Z., Zou, Y., Xie, B., 2019. CoRA: decomposing and describing tangled code changes for reviewer. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1050–1061.

Wettel, R., Lanza, M., 2008. Codecity: 3d visualization of large-scale software. In: Companion of the 30th International Conference on Software Engineering. pp. 921–922.

Wettel, R., Lanza, M., Robbes, R., 2011. Software systems as cities: A controlled experiment. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 551–560.

Zhang, T., Song, M., Pinedo, J., Kim, M., 2015. Interactive code review for systematic changes. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 111–122.

Enrico Fregnan is a Ph.D. student in the Zurich Empirical Software engineering Team (ZEST) at the University of Zurich. He received his bachelor's degree at Politecnico di Milano, Italy and his master's degree at Delft University of Technology, The Netherlands. His research focuses on investigating how to support developers during code review.

Josua Fröhlich is a Software Engineer at Business Systems Integration (BSI) Zurich. He obtained his Master's degree in Informatics with major in People-Oriented Computing at the University of Zurich in 2020. Prior to that, he received his Bachelor's degree in Applied Informatics in 2017 at the University of Zurich.

Davide Spadini is a Software Engineer at Meta focusing on Software Testing Infrastructure and Tools. He received his B.Sc. and M.Sc. in Computer Science from the University of Verona and Trento, Italy, and his Ph.D. in Software Engineering from Delft University of Technology. He focused his research activity in the software testing area. This included software testing practices, test code quality, test code review, and test code maintenance.

Alberto Bacchelli received the bachelor's and master's degrees in computer science from the University of Bologna, Italy, and the Ph.D. degree in software engineering from the Università della Svizzera Italiana, Switzerland. He is an associate professor of Empirical Software Engineering with the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland.