Universität Zürich[UZH]

Department of Informatics

Master's Thesis

Code Review Visualizations With CodeDiffVis for Java

Josua Fröhlich
13-732-268
josua.froehlich@uzh.ch

supervised by
Prof. Dr. Alberto Bacchelli
Enrico Fregnan
Zurich Empirical Software Engineering Team

30th June 2020

Abstract

Code review is a part of the software development cycle to improve code quality, the detection of bugs, knowledge transfer, and team awareness. Research in this field has focused on minimizing developers' effort while increasing their performance. An example is the creation of a changes ordering theory to reduce reviewers' cognitive load. However, code review can easily become a complex task, when the number of changes to review increases. We hypothesized that providing supportive figures such as a call or dependency graph would be helpful for a reviewer during code review. Therefore, we developed a tool to visualize the relationships among entities in the code to be reviewed. To evaluate our tool, we have conducted two qualitative studies: a user study with nine professional developers in a software development company and an online survey with 29 participants. In both studies, participants generally responded positively about the tool. Participants reported they were or would be able to understand the code changes quicker. Moreover, they found easier to navigate through the changes and to orient themselves in the merge requests.

Zusammenfassung

Code-Review ist ein fester Bestandteil des Software-Entwicklungszyklus, um die Code-Qualität zu verbessern. Zum Beispiel verbessert Code-Review Fehleridentifikation, Wissensaustausch und Teamgeist. Forschung auf diesem Gebiet konzentrierte sich darauf den Aufwand pro Entwickler für ein Review zu verringern und gleichzeitig mehr Nutzen daraus ziehen zu können. Ein Beispiel ist die "Change-Reihenfolge-Theorie", um die kognitive Belastung während eines Reviews zu reduzieren. Trotzdem kann Code-Review schnell zu einer komplexen Aufgabe werden, beispielsweise wenn sehr viele Changes überprüft werden müssen. Die darauf anknüpfende Hypothese war, dass eine grafische Unterstützung einem Entwickler beim Review helfen würde. Zwecks dessen wurde ein Tool entwickelt. Um das Tool zu evaluieren wurden zwei qualitative Studien durchgeführt, eine Benutzerstudie mit 9 Testpersonen und eine Online-Umfrage mit 29, respektive 67 Teilnehmern. Die Teilnehmer beider Studien waren von der vorgeführten Software mehrheitlich begeistert. Die Teilnehmer konnten schneller verstehen worum es bei einem Change geht. Darüber hinweg konnten sie einfacher zwischen den Changes hin und her wechseln und fanden sich allgemein besser im Code-Review zurecht.

Contents

List of Tables

List of Figures

1 Introduction

Modern code review is an established practice in software engineering to improve code quality, knowledge transfer and team awareness in both the industry and open source community [Bacchelli and Bird, 2013]. This was not always the case. Fagan who introduced the concept of code review, proposed it to be strict, formal and code-based [Fagan, 2002]. Since its introduction by Fagan, the process of code review has changed. There was a paradigm shift from its early adopted versions to a more lightweight and less rigid but tool-based approach [Rigby et al., 2012]. The reason for this shift was that, though code review was effective, it was not efficient [Baum et al., 2016]. For example, an effective code review detects bugs in the code change. An efficient code review does not waste unnecessary resources i.e. time. Baum et al. described modern code review as conducted in a peer-review style. Only the changed parts are reviewed by a member of the same software engineering team or by an expert of that particular system.

Today, code review is practiced with specialized tools. Despite these tools support a reviewer during a review, there is still potential to improve review effectiveness and efficiency [Baum and Schneider, 2016]. First, a software engineer with adequate expertise in the sub-system or framework under review needs to be selected. And second, he should not waste time with the review, i.e., to understand the code change. Both review effectiveness and efficiency can be improved with simple measures like sharing knowledge among the development team or adopting strong code style standards to improve readability [McIntosh et al., 2016]. However, there are limitations: E.g., a reviewer needs to know whether a code change part has been reused or if it is completely new. Furthermore, a reviewer has to find the optimal order for the walk through if there are multiple change parts to review [Baum et al., 2017]. According to Kononenko et al., higher change complexity and size, i.e., the number of lines of code changed, makes a review more challenging because a reviewer is expected to completely understand the code change in its greater context [Kononenko et al., 2016].

In the past years, research has focused on improving the process of code review. They investigated various solutions to improve the aforementioned issues. Some solutions try to minimize the amount of work and improve the review effectiveness by adopting automated code reviews, to automatically identify software defects and enforce code style standards [Baum and Schneider, 2016]. An example of such an application is SonarQube[1], suggesting improvements while a programmer is actively coding in his integrated development environment (IDE). For code reviews, change parts are usually ordered alphabetically. Baum et al. have considered the problem of *optimal ordering of review changes*, introducing a new metric that groups related change parts. This allows constructing a graph and to select the most relevant change part first [Baum et al., 2017]. Other applications focus more on graphical interface enhancements such as GitHub[2] or its open-source variant GitLab[3]. The graphical support makes it more clear what change has been made and in which context. They make it easier to detect whether a code part is deleted, changed or added brand new.

A problem with graphical tools like GitLab is that they do support code review in a lightweight manner only. Such applications are limited if, i.e., a code change stretches over dozens of files. As a result, a reviewer may lose the overview. While the solution from Baum et al. proposes a better ordering than alphabetical ordering, reviewers nevertheless may have to jump back and forth between changes.

Baum and Schneider underscored also the need for support for understanding code changes. We hypothesize that, even for code changes in a handful of files only, providing a visual overview of the current changes would be beneficial. For example, reducing the time required to understand a review or improveing knowledge transfer. For complex code changes, reviewers have to

[1] https://sonarqube.org
[2] https://github.com
[3] https://about.gitlab.com

construct a dependency graph or a call graph in their head to grasp the information flow within the change. As the dependency graph and the call graph is contained in the source files' abstract syntax tree (AST) such a graph can be generated automatically for a code review.

We developed a tool that uses a graph visualization to display class dependencies and method calls among the changes in a code review. We evaluated the usefulness of the tool in two qualitative studies. First, a user study with professional developers in the Swiss company BSI was deployed after preceding pilot study was conducted in the same company. The feedback was collected with a survey. And Second, an online survey was distributed over social media platforms to gather a broader feedback on the concept of the tool. These participants where introduced to the tool with explaining images and videos. In both studies, participants responded rather positive about the proposed solution.

2 The Problem

The long-term problem targeted in this thesis is to improve effectiveness and efficiency of modern code review by reducing the cognitive load on the reviewer. One of Baum and Schneider's findings was that the need for tool support was evident. Though they emphasized that it should not impair the reviewers flexibility to review the code in their own way [Baum and Schneider, 2016].

Baum and Schneider concluded their study with four takeaways:

> 1. Code review effectiveness and efficiency depend to a large degree on the reviewer, its style of work and its fit to the artifact under review.
> 2. Understanding the review artifact is the most important aspect of reviewing code.
> 3. Review in industry is commonly done change-based.
> 4. The review of large changes is the most significant challenge in code review.
>
> [Baum and Schneider, 2016]

Based on these implications, we believe the following thoughts emphasize on the need for tool support. Firstly, one issue with substantially large changes is that a developer has to keep track of all change parts already reviewed and the ones still requiring a review. A large change involves, i.e., more than seven files. And secondly, to understand the review artifact, source code alone may sometimes be not sufficient. The reasons could be various. The reviewer may be unfamiliar with the type of code or the used frameworks and concepts. In this case they might not be the best fitting reviewer. Researchers focused on improving reviewers' assignment: e.g., Yu et al., among others, proposed a solution by mining each project's comments [Yu et al., 2016]. Even if the best fitting reviewer was found, it takes some time to familiarize with the code, not necessarily just because the change is considered *complex*.

2.1 The Challenge

It is a challenging task to develop a tool to improve a reviewers effectiveness and efficiency. One reason is that it should not interfere with the flexibility of the reviewers to perform code reviews in their personal style. For example, Baum et al. found evidence that the current order in which review tools display code changes is far from optimal [Baum et al., 2017]. Enforcing a close-to-optimal change order still leaves many issues open: e.g., jumping back and forth between change parts or losing track of which parts have already been reviewed. Another reason why this problem is difficult to solve is based on the fact that review style changes with each reviewer and is not bound to a fixed procedure. Therefore, tools that might be useful to some reviewers might not be considered effective in general.

2.2 Example

To understand a review artifact, a reviewer needs to be either familiar with the type of code or with the used frameworks, or a reviewer should be able to easily learn how they work. During a review, he needs keeping some abstract representation of the classes and methods in his mind, such as a call graph or a dependency graph, to fully understand the greater picture. A reviewer has to search for class names and method calls using the search option from the tool or the browser in case of web-based tools which leads to frequent navigation within the change code. We address the problem by supporting the reviewer with informative visualizations. Method calls and class dependencies can be extracted from source code and visualized in the browser.

2.3 Thesis Statement

The thesis statement is as follows:

> Visualization techniques support a reviewer during a code review. A useful visualization would be one that displays method calls and dependencies among classes that are part of the change.

To validate our thesis, we developed a tool, *CodeDiffParser* that analyzes code change of Java source code and generates an abstract syntax tree. The output is consumed by an extension for Google Chrome, called *CodeDiffVis*. It hooks on GitLab merge request pages and draws a call and dependency graph. The visualization is interactive and customizable. We evaluate the usefulness and benefits of the developed software through a user study with an online survey. Preceding to the user study, a pilot study was be deployed. The intention was to gather general feedback on whether the primary idea fits an actual gap in developers' need of tool support for code review. The user study, employing an updated version of the tool, additionally included an online survey. Both studies where conducted with professional developers in their personal working environment. Participants where advised to utilize the tool for at least one code review before responding to the questions in the survey. Additionally, an online survey was distributed via social media platforms to reach a broader mass to obtain more diverse results. We validate our hypothesis analyzing the answers from both the user study and the online survey.

3 Background and Related Work

In this section, we address the problem in its greater context and related research. In Section 3.1, a deeper look into how source code is typically produced and visualized during development, in IDEs is given. Similarly, for code review, the way code change parts are visualized is described in Section 3.2. In Section 3.3 we provide examples of how information such as the AST or dependencies can be extracted from source code. These techniques will be used for the implementation to construct call graphs or dependency graphs. Related tools or tools that use the same or similar techniques are explained in Section 3.4. Moreover, we provide an explanation why we believe these tools do not fully cover the problem stated earlier in Section 2.

3.1 Source Code Visualization

In modern integrated development environments (IDEs) like the Eclipse IDE or IntelliJ IDE, source code is typically presented file-based. In object-oriented programming (OOP), each file contains a class or interface which further contains features. The problem with this approach is that display space is wasted. One reason is that files contain lots of white space. Another problem is that often code which is irrelevant to the developer is displayed. As a result, frequent navigation in code is necessary to keep all relevant information in a developer's mind. Many researchers focused on the aforementioned problems [Plumlee and Ware, 2006][Bragdon et al., 2010b][DeLine et al., 2012]. Bragdon et al. introduced a new metaphor called *bubble*, which is a editable and interactive view of a fragment such as a method or collection of member variables. The idea behind a bubble is, that a developer only sees and reads the relevant part of code instead of the whole file. This makes navigating through code more efficient. In another study by Bragdon et al., in a quantitative experiment, they found evidence that their *Code Bubbles IDE* significantly reduced the time required for navigation and the time developers need to understand code [Bragdon et al., 2010a].

Another possible representation of source code is using graphs. Nodes represent files (or code change parts) and edges represent relations between nodes. Relations can be call based, i.e. method calls from class A (node 1) to class B (node 2), resulting in a *call graph*, or dependency based, where relations represent dependencies among software entities: e.g., object creation from class B in class A or inheritance between the nodes. Such a graph is also feasible for only pieces of code, i.e. a code change for a code review. The approach of constructing a graph based on the relatedness of code changes was introduced by Baum et al. in 2017 and is explained in more details in Section 3.4 [Baum et al., 2017].

3.2 Code Collaboration, Version Control and Code Review

Since its introduction by Fagan in 1976 [Fagan, 2002], code review is part of the source code development cycle. The code review by Fagan was very time and resource consuming. Since its early adopted versions, code review has changed from being structured and formal to unstructured and informal [Bacchelli and Bird, 2013]. Modern approaches are typically change-based, lightweight and are supported by tools. The term *Modern Code Review (MCR)* was introduced by Bacchelli and Bird to emphasize said paradigm shift [Bacchelli and Bird, 2013]. MCR is performed in a peer review style such that the changed code parts (either pre or post commit), are reviewed by at least one developer, the *approver*, different from the original developer, the *author*. Reviewers may approve or leave feedback in form of comments. In the comments, an approver mentions i.e. bugs that need to be fixed by the author. Again, these changes have to be reviewed. Code review has several advantages, not least the detection of defects in the source code. Bacchelli and Bird have found more advantages, such as improved knowledge transfer and team awareness and the creation of alternative solutions.

Many tools (both commercial and open-source) exist to support code review. Well-established tools are based on the versioning tool Git[4]. Some well-established tools are GitHub[5], hosted by Microsoft or GitLab[6], a similar open source platform. While providing source code versioning management, these tools also offer issue tracking and continuous integration and development (CI/CD) pipeline features. Pipelines can run e.g. unit tests on the code change in advance of the code review. The review is done on a *git diff* based view, that shows the added and deleted source code parts. Comparing two source code bases, so called *branches*, is called a *merge-request* or a *pull-request*. Figure 1 shows schematically a merge-request creation in GitLab. There are different strategies to unite two branches. *Merging* two branches means that the source code bases are compared as is, whereas *rebasing* tries to preserve the changes in historical correct order for both branches. The approver needs to review the differences in all files from both branches only.

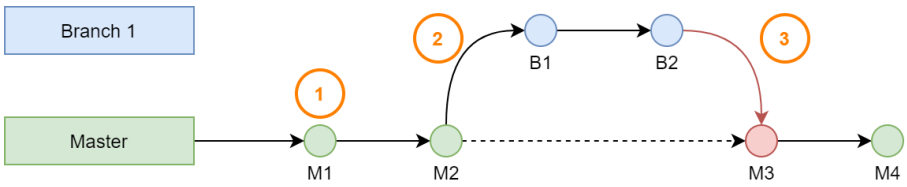


Figure 1: Git-based system using two branches. 1) Commits on a branch are denoted with a circle and a commit ID. 2) *branch 1* is created from branch *master* with the code base of commit *M2*. 3) After the work on branch 1 has been completed with a final commit *B2*, a merge request is opened in GitLab. If the merge request is approved, a new commit *M3* on the master branch is created as branch 1 is merged to master.

Other web-based source code versioning management tools are Gerrit[7] and Phabricator[8]. Gerrit is a open-source project, initially developed by Google and its main strength lies in code review support and tracking commit changes. Unlike Phabricator, Gerrit does not provide any issue tracking. Phabricator was initially developed by Facebook but is now available open-source. Besides Git it supports SVN and Mercurial versioning control systems.

3.3 Abstract Syntax Tree and Dependencies

The following steps are required to generate an abstract representation of source code, e.g. a graph. First, the source code has to be parsed to construct a hierarchical representation. From that the required nodes are collected, e.g., classes or methods. Dependencies of classes or methods can be used to draw links between these nodes. In the following two sections we explain first the concepts of producing an abstract representation of source code in Section 3.3.1 and second the types of dependencies that can be extracted from Section 3.3.2.

3.3.1 Abstract Syntax Tree (AST)

Here we provide a brief introduction to the theory of AST parsing. Moreover, we demonstrate an AST with an example of a parsed source code snippet using Eclipse JDT AST parser. Finally, we mention other frameworks that can create AST from Java source code. Creating an AST from Java byte code is not meaningful because the byte code would require to be decompiled first.

An abstract syntax tree (AST), sometimes called syntax tree only, is a hierarchical representation of parsed source code. It is different to a parse tree because it exclusively contains the

[4]https://git-scm.com
[5]https://github.com
[6]https://about.gitlab.com
[7]https://gerritcodereview.com
[8]https://phacility.com/phabricator

nodes relevant for further analysis [Cooper and Torczon, 2011]. Information such as brackets or comments may not be considered relevant, e.g., if the AST is used for a compiler. One framework for parsing source code and to construct ASTs is Eclipse's Java Development Tools (JDT) Core Component[9] introduced by the Eclipse Foundation. It is intended to use for plugin development for the Eclipse IDE, but a standalone variant using an arbitrary artificial workspace is equally possible. Given an environment like the Eclipse IDE workspace, Eclipse JDT can modify contained source code files or resolve its dependencies during the parsing process [Aeschlimann et al., 2005].

```
1 package test;
2
3 public class SampleClass {
4   public void foo(String bar) {
5     bar.toUpperCase();
6   }
7 }
```

Listing 1: Simple example of a Java class. It's AST is shown in Figure 2.

Consider Listing 1 with a simple Java class parsed using Eclipse AST parser. Even for straightforward cases, the AST can grow arbitrary complex. Provided that all necessary dependencies are available at parsing time, the AST can also resolve method or type bindings to resolve class or method level dependencies. The resulting AST of analyzing the short code snipped is schematically drawn in Figure 2.

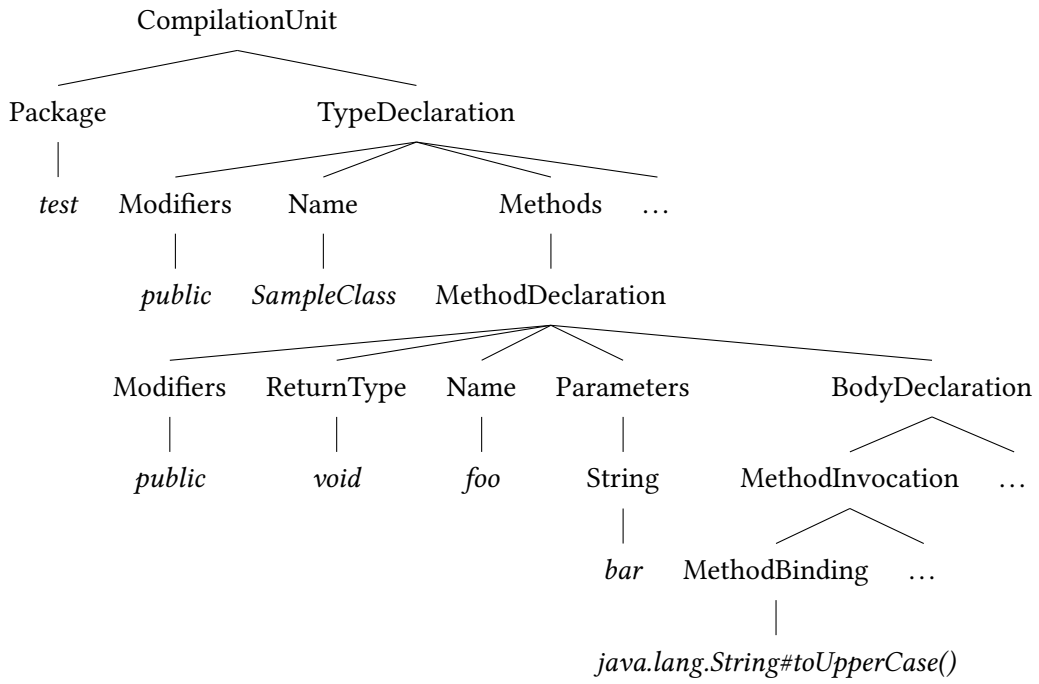


Figure 2: Visualization of a simplified AST for Listing 1 using Eclipse's Java AST Parser with a hierarchical AST visitor. For demonstration purposes the graph is drastically simplified. Each node represents an object and the member variables of a node represent their children or leafs. Leafs are shown in their string representation in italic. If a node has more member variables than shown, it is indicated with a "…" symbol. Method bindings can be resolved already while the AST is parsed node by node. In the figure, the method binding, i.e. method call of `String#toUpperCase()` from `SampleClass#foo()` is already shown in its resolved form.

[9] https://eclipse.org/jdt/core/index.php

There exist other AST parsers for Java source code such as JavaParser [van Bruggen et al., 2008] and IntelliJ's AST parser[10]. JavaParser is a framework introduced in 2008 and was primarily targeted for AST creation and live source code manipulation. Overtime they additionally included a dependency resolution mechanism, called SymbolSolver. The problem with JavaParser's Symbol Solver is that it is not equally mature compared to the dependency resolution mechanism from Eclipse's AST parser. This is primarily because it is a new framework with a smaller user-base compared to Eclipse's JDT. IntelliJ's AST parser is intended for solely extending IntelliJ IDE. Eclipse JDT, despite using pre Java 7 syntax, has been found to be more robust against new API versions compared to JavaParser and IntelliJ's AST parser.

3.3.2 Coupling and Dependencies

Coupling and dependencies represent a key concept in OOP. To draw the links between the extracted nodes in the AST, a coupling analysis needs to be performed to reveal the nodes' dependencies. In the following paragraph we explain this concept in more detail.

In OOP, dependencies are usages of an object in another object. E.g., a class (or interface) *B* is used by a class *A*. If there is a dependency between two classes, the two classes are *coupled*. Coupling relations can be divided into four subclasses [Fregnan et al., 2019]:

1. *Structural coupling*. Entities that have static relations in the source code such as inheritance or method references
2. *Dynamic coupling*. Entities that are related through dynamic binding or polymorphism. This can only be addressed at runtime
3. *Semantic coupling*. Entities that are related because of their comments or identifiers. Semantic coupling between entities is extracted using machine learning techniques
4. *Logical coupling*. Entities that belong together logically, are entities that, i.e., are frequently changed together

Similarly as for coupling, dependencies can be *static* or *dynamic* [Jenkov, 2014][Tate and Gehtland, 2004]. Here, the former is referring to dependencies that can be resolved at compile-time, the latter can only be resolved at runtime. Parsing an AST may at most reveal static dependencies. Three unique kinds of dependencies are formulated by Jenkov and are resumed in Table 1 [Jenkov, 2014]. It is substantial to notice that Jenkov makes a difference if there is a method dependency, from a method of one class to a method of another class or an actual class dependency. Unless the method dependency is on a static class, there is always a class or interface dependency as well.

Dependency type	Explanation
Class dependency	A class or interface that has a dependency on a class. This can be a class hierarchy dependency or a class reference, i.e., an instantiated object of another class.
Interface dependency	A class or interface that has a dependency on an interface. This can be a class implementing an interface, an interface that is extending another interface or a type reference.
Method dependency	A class or interface that has a dependency on a concrete method or field of an object.

Table 1: Dependency types in OOP as described by Jenkov [Jenkov, 2014]. In Java, interfaces can not be instantiated. At runtime, an interface dependency is always on a class that implements that interface. A similar argument can be applied for abstract classes.

[10] http://jetbrains.org/intellij/sdk/docs/welcome.html

3.4 Related Tools and Concepts

In this section, tools that use the same or similar of the aforementioned concepts are explained. With each tool, a brief analysis is provided why we think they do not adequately cover the problem described in Section 2.

3.4.1 JDeps

For dependency analysis of Java projects, Oracle's in-house product JDeps[11] already comes with a tool that can draw simple dependency graphs. JDeps analyzes Java byte-code *class* files and resolves the static dependencies between classes. The tool is part of the Java development kit (JDK). Although it provides some configurability such as resolving the dependencies on package or class level, or filtering out archive dependencies, it does not parse source code and is thus unsuitable to process dependencies on code changes or detect line numbers with dependencies.

3.4.2 Change Part Ordering

Baum et al. developed the theory of optimal code change ordering by grouping related change parts [Baum et al., 2017]. Relatedness is extracted from the underlying call graph of the whole change to reduce the cognitive load on the reviewer. But there are also other possible relations between two change parts: e.g., inheritance, or if the changes occurred in the same file. Although employing the theory would reduce the amount of navigation while reviewing code, therefore reducing developers' cognitive load, we believe the cognitive load could more significantly be reduced by introducing visual support to display the relations.

3.4.3 Softagram

Softagram[12] analyzes source code and its dependencies based on a merge requests. It supports multiple platforms including GitHub, GitLab, Bitbucket, Azure and Gerrit, as well as various programming languages (among others C++, Java and Python). Upon a merge request, Softagram triggers an analysis and posts the result as a SVG graph in the merge request comments. The graph includes added and removed dependencies, and hidden or unwanted dependencies. Changed components or classes in a merge request are highlighted and the incoming and outgoing dependencies of each package and class can be shown. For more detail, Softagram desktop, a standalone application to navigate the graph, reveals the data at a customizable detail level, i.e., on package or class level. Softagram focuses on dependencies only, and provides a rule set that simplifies detection of erroneous dependencies. As a drawback, call graphs cannot be produced and the generated dependency graphs become quickly complex.

3.4.4 RoleViz

Similar to Softagram is RoleViz, a tool developed by Ho-Quang et al. for whole project inspection [Ho-Quang et al., 2019]. Unlike Softagram, they adopt a alternative approach for the visualization: components for each Java file are divided into *roles*. The concept of roles was introduced by Wirfs-Brock in 2006. The idea is that each piece of code receives a role with a responsibility. Wirfs-Brock classified 6 different roles. For a class representation, RoleViz uses the most common role found within that class as the main role. Likewise classes, packages are analyzed for dependencies among each other. RoleViz was compared to Softagram in a study conducted by the same author and has been found to be equally powerful comparing complexity, cognitive load and understanding [Ho-Quang, 2019]. RoleViz was not tested on a code change basis, but only for whole projects.

[11] https://docs.oracle.com/en/java/javase/11/tools/jdeps.html
[12] http://softagram.com

4 Proposed Solution

In Section 4.1, we describe the tool we developed to visualize code changes for review in more detail. We developed the visualization tool, *CodeDiffVis* (CDV), that visualizes a call and dependency graph of the code change, described in Section 4.1.2, and a source code analyzer, *CodeDiffParser* (CDP), that parses Java source code, i.e., the code change and generates the input Json for CDV, described in Section 4.1.1. The two tools operate independently of each other. CDV uses the output of CDP as input to produce the visualization, but CDP could be exchanged with any other parser that produces an output that matches the application programming interface (API) of CDV. We further provide a list of the implemented features and the possible interactions with the tool in Section 4.1.3, along with an example in Section 4.1.4. We finalize this section with the limitations of the tool in Section 4.2.

4.1 Design and Implementation

Baum and Schneider analyzed the need for tool support for code review. They found multiple potential ways how a reviewer could be supported by tools to help understand the code changes better. Among others, they mentioned "helping the reviewer to understand large changesets" and concluded that in this context, a reviewer could benefit from a summary or a visualization of the code change. Therefore, one of the core aspects of this work was to develop a tool that supports a reviewer visually during a code review. For this thesis, the scope of the tool had to be defined initially. For example, we decided that the programming language Java would be sufficient at first and that CDP could be extended from there. Since Java is a well established OOP language that has a vast popularity, we decided to concentrate on Java projects only [Carraz et al., 2019]. For the visualization, it seemed best to start with support for GitLab as it is open-source and commonly used in practice by developers [13]. Additionally, both Java and GitLab are used at the company where we conducted the user study.

For the *iterative design and implementation phase*, a first challenge was to find a suitable framework to parse Java source code. Parsers typically construct an abstract syntax tree (AST) from source code and resolve method and class dependencies from compiled code. Depending on the implementation details and complexity, the best suited framework had to be chosen. We explored three following frameworks:

1. *JavaParser*[14]. Creates the AST along with its SymbolSolver that resolves dependencies. It's the newest framework in this list and is still actively in development
2. *Eclipse AST Java parser*[15]. The framework from the Eclipse Foundation, which is written in a bit stale Java style but supports all Java versions
3. *IntelliJ's AST parser*[16]. Is written in modern Java style but was also developed solely for IntelliJ plugin development

Although JavaParser was convenient to use, we found its SymbolSolver was not mature enough to resolve all dependencies correctly. For that reason JavaParser was dismissed. IntelliJ's AST parser was hard to run in a standalone example without an IntelliJ project. We encountered further problems with its dependency solver especially with more recent versions. In addition, the version updates differ heavily in how to write the plugin which was the primary reasons why this framework had to be dismissed. Eclipse AST Parser was intended for Eclipse plugin development but could be run in a standalone application with some workarounds. We found its

[13] According to GitLab's website, more than 100'000 organizations utilize the software. https://about.gitlab.com/
[14] `https://javaparser.org`
[15] `https://www.eclipse.org`
[16] `https://www.jetbrains.org`

API did not change over the Java versions 8 to 13. Furthermore, we obtained the best results for dependency resolution in our example project.

A second challenge in the design process was to identify a suitable way to display the graph for code review. The initial design mockups for the back-end and front-end, respectively, can be found in Appendix A. An example of the planned architecture setup can be found in Appendix A Figure 11, a sample class visualization with method calls in Appendix A Figure 12 and a sample merge request design in Appendix A Figure 13. We found it would be straightforward to develop a browser extension that uses the analysis from the back-end and just visualizes the data. The reasons are various. First, a browser extension is less invasive than, i.e., a GitLab plugin and therefore the acceptance might be higher to install the tool for productive use. Second, the visualization is kept separate from the GitLab source code and there are no side-effects, i.e., being less dependent on GitLab versions. And third, a reviewer may always decide whether to use the tool, or else, to disable the extension. If the tool was part of GitLab, this would be more complex to accomplish.

4.1.1 Back-End: CodeDiffParser

CodeDiffParser can be used as a back-end component of CDV. It describes the API, i.e., what format the Json needs to be so CDV can consume it and draw the visualization. CDP uses the `HierarchicalASTVisitor` from the Eclipse AST parser to construct an AST, then traverses that tree top to bottom. An example of such a tree was provided in Figure 2. CDP analyses type declarations, method declarations and type instantiations for the dependencies and method invocations for the call graph that will be displayed with CDV. The resulting tree captures the relation between nodes via the links. For example, if a method belongs to a class, there is a link with a relation `METHOD`. On the contrary, if there is a method call from one method to another method, there is a link with a relation `METHOD_CALL`. Nodes are marked by whether they are simply referenced or part of the analyzed code change.

For both branches, the source and target branch of a merge request, the same analysis is run and the resulting graph is compared node by node. First, the target branch (i.e. the branch to be merged into) is analyzed and all nodes and links are marked as `DELETED`. In a second run, the source branch (i.e. the branch to be merged from) is analyzed and if a node is already present in the tree, it is marked as `UNCHANGED`. If the node was not present in the first tree, it is added and marked as `ADDED`. After the analysis, all parents in the hierarchy of added methods or inner classes nodes are marked as `CHANGED`. The result is saved in a Json format, where all nodes and links are stored in two lists, accordingly.

4.1.2 Front-End: CodeDiffVis

The Google Chrome browser extension, *CodeDiffVis*, downloads the aforementioned Json either from a local storage or a web service such as Jenkins. The D3Js library is used to produce 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. Due to the typology of the force-directed graph, each node is attracted to the center of the graph while being repulsed by other nodes. An auxiliary force keeps connected nodes in their near surroundings. D3Js' graph layout recalculates the SVG each tick until an alpha threshold is surpassed. Nodes do not overlap. The graph SVG is opened in a separate window allowing the reviewer to display it on a second screen. The source code displayed in GitLab and the graph are connected through the features described in Section 4.1.3. An example of the SVG is demonstrated in Figure 3.

CodeDiffVis additionally provides settings, so a reviewer can adjust the graph to his needs. The settings are stored locally such that a reviewer can reuse the same settings for all merge requests. An example of the settings view is provided in Figure 4. The following settings are implemented:

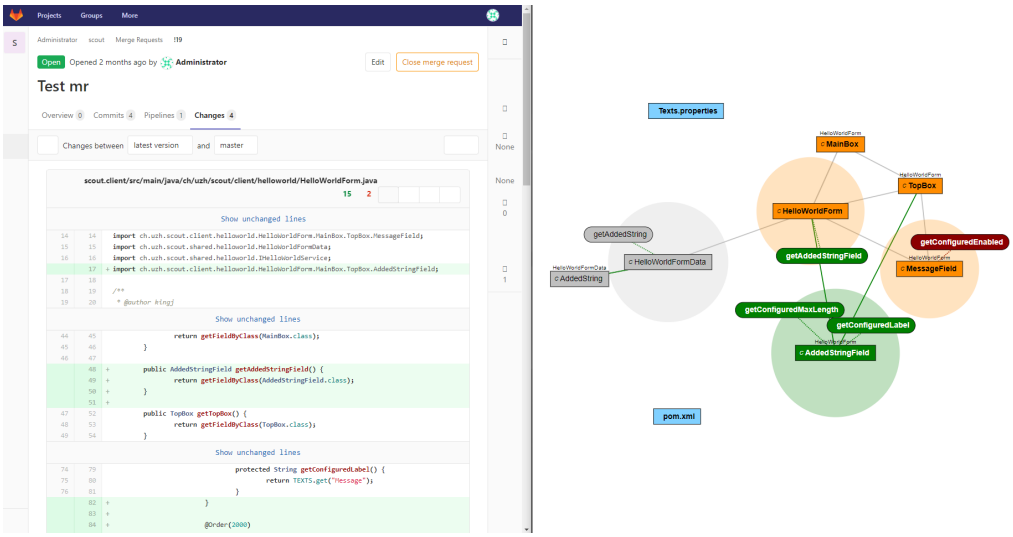


Figure 3: Example of how the tool integrates the graph into a GitLab merge request page using content scripting (JavaScript injection).

1. *Path to the Json file.* Allows to set the path to the generated files, i.e., by CDP. The path is either relative to the location of the extension (local storage) or as a web url. If no file name is specified, it assumes the merge-request ID as the file name.

2. *Color schema* toggle. Switch between the two implemented color schemes. Nodes can either be colored by change status or by package. For packages, there is a rainbow color-map used that is adjusted to the number of packages displayed. For change-based coloring, there are six colors used: green (added), orange (changed), red (deleted), grey (generated nodes: changed, added or deleted), white (unchanged) and light blue (non-Java nodes).

3. Show or hide *non-Java nodes* toggle. Show or hide nodes that are labeled `NON_JAVA`.

4. Show or hide *generated nodes* toggle. Because generated classes and methods typically do not require code-review, but may be relevant to display all the nodes' dependencies.

5. Show or hide *methods* toggle. If a reviewer is just interested in all changed classes but not in their methods in general.

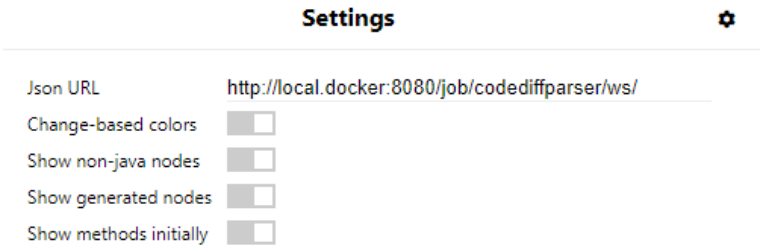


Figure 4: The settings view of the Chrome extension.

Based on the feedback from the pilot study, we decided not to integrate the graph directly in the GitLab page, which is also different from Softagram's approach of adding a comment in the merge request that displays the dependency SVG. Furthermore, we decided against an encapsulated layout of classes similar to how it is utilized in the unified modelling language

(UML)[17], the most widely used standards for specifying and documenting information systems [Fuentes-Fernández and Vallecillo-Moreno, 2004]. Different from UML, we extracted the methods from their classes and used a complete flat layout instead. Methods are connected with links to their corresponding class or interface. We argue it would be straightforward to draw and understand a graph with many nodes and connections between methods and classes with the flat layout, and that it ensures all connections are clear. Another reason for the flat layout was, due to the used framework for the examples; Eclipse Scout[18]. Scout uses many nested classes which would clutter the display, and it would become complex to distinguish which nodes are connected if the nodes are encapsulated. We adopted that framework because it is an open-source product of the Swiss company *Business Systems Integration AG* (BSI), where the pilot study and the user study were deployed. On the contrary to our approach, RoleViz and JDeps, as described in Section 3.4.4 and 3.4.1, used a module or Jar based approach, which we believe would not fit for a code review where a reviewer has to analyze changed methods, for example.

4.1.3 Interactions

In this section, we describe the ways in which reviewers can interact with the graph, e.g., to amend the layout or to track his progress during the review. Connected nodes can be added and removed, such that the graph is arbitrary customizable to the reviewers' needs. There are four kind of interaction categories:

1. *Layout.* Alter the appearance of the graph, i.e., by placing nodes differently, resizing the graph or the whole visualization or highlighting sub-graphs of connected nodes

2. *Navigation.* Track the progress of the review and link the graph to the code on the GitLab page. Clicking on nodes jumps to the position in the source code

3. *Reverse Navigation.* Hovering over the source code in GitLab highlights the corresponding node in the graph. The graph automatically adjusts zooming and centers the highlighted node

4. *Customization.* Customization for the graph by adding or deleting of nodes. For each node, all connected nodes can be added, i.e. referenced nodes. Nodes can be deleted i.e., if a reviewer has finished with reviewing that particular node

All possible interactions are described in Table 2 in more detail.

[17] `https://www.uml.org/`
[18] `https://www.eclipse.org/scout/`

Name	Category	Explanation
Pan	Layout	Pan the whole graph around to explore various parts.
Drag	Layout	Drag nodes around and adjust their positions.
Zoom	Layout	Zoom in and out of the whole graph, i.e., to see the greater picture or to emphasize on a sub part of the graph.
SVG resize	Layout	Resize the whole SVG as it can overlap with the source code displayed in the review.
Hover	Layout	Hover over a node to highlight all its nearest neighbors. This is especially beneficial to reveal subordinate call graphs.
Hover reverse	Reverse Nav.	Hover over the code in GitLab to highlight a node with all its nearest neighbors. The graph automatically centers the hovered node.
Hover lock	Layout	Lock and unlock the current highlighting of the graph.
Link	Navigation	On clicking on a node, jump to the first occasion in the code change directly. When a node was clicked, it alters its appearance to make it easier for a reviewer to track which nodes were already reviewed.
Expand	Custom	Show any connected (sub-) nodes i.e. referenced nodes.
Remove	Custom	Remove any node that is not necessary (anymore).

Table 2: Possible interactions with the graph. The idea is a reviewer can adjust the graph to adjust it for his review style.

4.1.4 An Example Review With CDV

Suppose a code change of a new feature is being analyzed. Based on the differences in both branches, the corresponding call and dependency graph is created and visualized in GitLab. In Figure 5 a graph from a sample merge request is shown and explained.

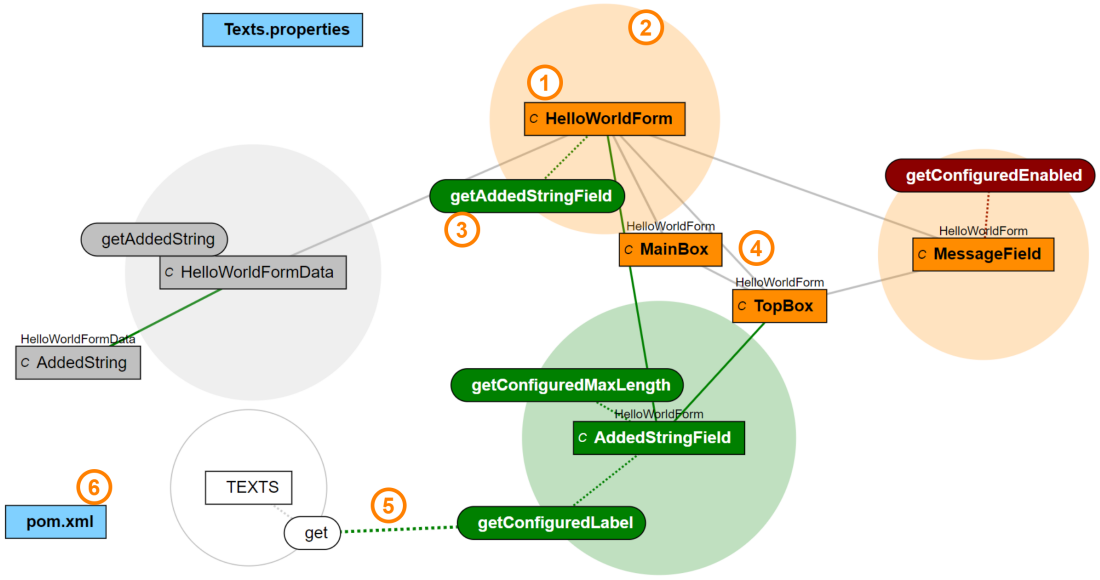


Figure 5: Example of a generated graph for a code change. Taken from a sample merge request from an Eclipse Scout project.

The sample change consists of three files, of which two are not Java files (XML files). The color coding for change-based coloring for nodes and links, as shown in Figure 5, is the following:

Added (green), changed (orange) and deleted (red). Generated nodes and links are grey, while unchanged ones are white. Non-Java nodes are pale blue. We further explain the graph layout in six steps while referring to Figure 5:

1. Rectangular nodes represent Java classes. Classes are denoted with C, Abstract classes with A and Interfaces with I
2. A circle around classes is added to nodes that contain at least one method that is part of the code change
3. Method nodes are shown as rectangles with rounded corners
4. Inner classes are denoted with a label on the top with the corresponding class file name
5. Dotted lines are used to represent method calls. E.g., Method `getConfiguredLabel` of class `AddedStringField` calls method `get` of class `TEXTS`
6. Nodes that do not represent Java components are shown with their full name

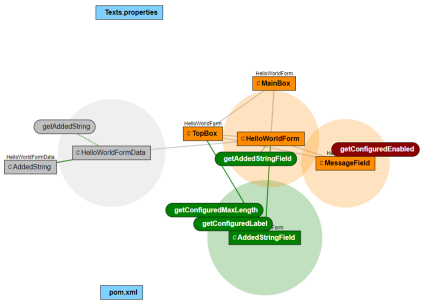
Hereafter, in Figure 6 a possible walk-through for a code review using CDV is provided. The snapshot illustrations for each stage where taken from the same sample merge request.

4.2 Tool Limitations

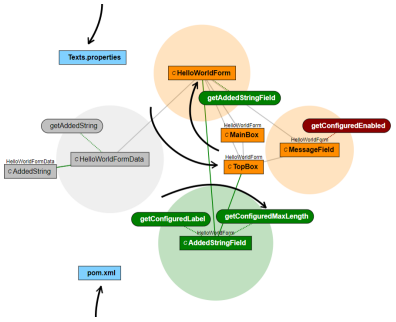
Some of the design choices where explained in Section 4.1. Because of these choices, and, since CDP and CDV had to be fit for the pilot study and the user study in the Swiss company BSI, some trade-offs and limitations were inevitable. Other limitations were due to the limited scope of the thesis. We present a non-conclusive list of limitations:

1. *Programming language.* CodeDiffParser only supports Java. To support another programming language, a separate parser needs to be implemented. The API provided enables CodeDiffVis to work independently from the programming language
2. *Supported Frameworks.* Only GitLab 12 is supported because it was the most recent version of GitLab during the implementation phase. Other versions were not tested, but could possibly work as well.
3. *Supported Browsers.* CodeDiffVis only supports Google Chrome because it is currently the most widely used web-browser according to StatCounter[19] and NetMarketShare[20]
4. *Transitive Dependencies.* If there is a dependency from object A to object B, and object B has a dependency to object C, then there is a transitive dependency from object A to object C. Transitive dependencies are not visible in the graph, i.e. if object B was missing, no link between A and C would be drawn
5. *Comment only changes.* CodeDiffParser analyzes the AST of Java source code. Comments are not contained in the tree, therefore comment only changes won't be recognized as changes in a review

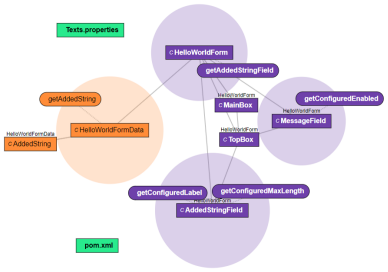
[19] `https://gs.statcounter.com/browser-market-share`
[20] `https://netmarketshare.com/browser-market-share.aspx`



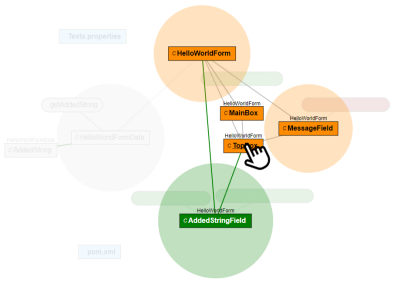
(a) This is the initial graph layout when the GitLab merge request page loads



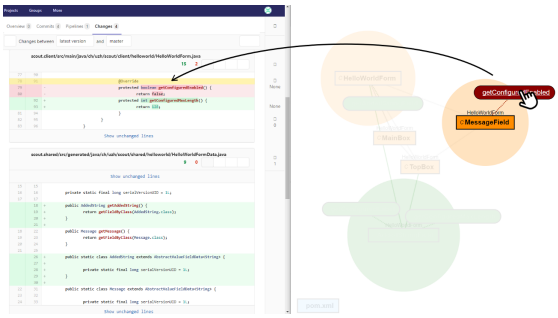
(b) Some nodes where rearranged manually via dragging as indicated through arrows



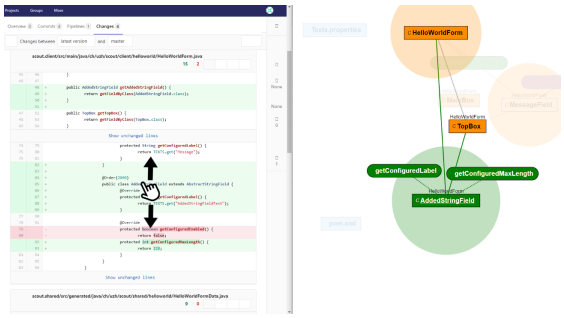
(c) Switching to package-based coloring reveals that most classes belong to the same package



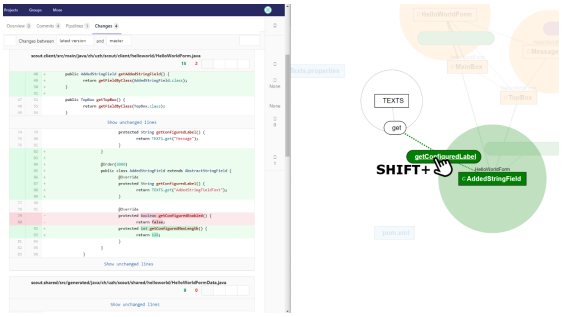
(d) Highlighting the TopBox node and its connected neighbors through hovering over with the mouse



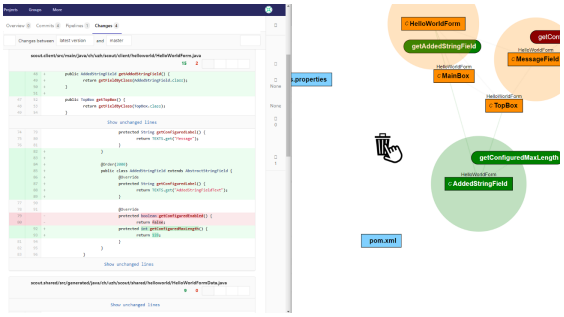
(e) Click on a node, i.e., a method to jump to its definition – the line in the source code – in GitLab



(f) Hovering over the source code in GitLab highlights the hovered node and its connected neighbors



(g) Shift + click expands, i.e., method and shows also referenced methods and classes



(h) Nodes, i.e., when finished with reviewing, can be deleted

Figure 6: Example walk-through on a sample merge request using CDV.

5 Evaluation

In this section, we explain our evaluation methods and results from the pilot study, the user study and the survey. Based on the problem description in Section 2, in Section 5.1 we define three research questions that divide the greater picture into simpler problems. In parallel to the user study described in Section 5.2, a online survey was deployed on a larger scale, as described in Section 5.3. For the user study and the online survey, the feedback from the pilot study was incorporated to improve both. We explain our methodology along with possible threats to validity for both, the pilot study and the user study in Section 5.2, and the survey in Section 5.3. In both sections we present the results accordingly in Section 5.2.3 and Section 5.3.3.

5.1 Research Questions

The principal goal of this work is to provide a proof of concept for enhancing light-weight and web-based code review tools for Java such as GitLab with visualizations that support a reviewer in code reviews. Because this work is restricted to Java projects, it is crucial to not rely too much on concepts for Java itself but more generally on OOP concepts. Suitable frameworks to extract the information have been described in Section 3.3. The rationale was to develop an application programming interface (API) that is easily adoptable for other object-oriented languages and provides a standardized format for the visualizations. Moreover, we want to find out what makes a call or dependency graph visually appealing and at the same time informative such that it would be useful for code review. To assess this question, the tool was provided to professional developers who utilize it in either their private or a productive environment, and, to evaluate the tool conceptually, it was distributed over various social media platforms. It follows our two research questions along with an explanation and justification for each.

> **RQ1** What is a reviewer's perception of CodeDiffVis as a means to support code review in a productive working environment?

We want to answer this question with a user study and deploy the tool in a productive environment. In a preceding pilot study, the primary goal was to assess whether the tool and its functionality appeal professional developers in general. For this user study, feedback from developers that use the tool for code review for productive applications is gathered in an online survey and semi-structured interviews. An improved version of the tool as in the pilot study will be distributed to gather a broader and more feedback. We argue it is necessary to make the testing of the tool as close to a developer's productive working environment, because only then a developer can judge the usefulness of the tool in practice.

> **RQ2** What is a reviewer's perception of CodeDiffVis as a means to support code review when evaluated as a concept?

This question is addressed with a survey, deployed on a larger scale that is spread via various social media platforms, i.e., Twitter, Reddit and WhatsApp. Example images and videos will be used to illustrate the functionality of CDV. Therefore, the updated version from the user study will be used along with a sample merge request. In the survey, participants are asked to evaluate the tool and the visualization techniques based on images and videos of CodeDiffVis using a sample merge request. We argue that with this second study we will be capable to assess whether CDV sustains the critique and understanding of the tool conceptually of a broader user base. I.e., not only professional developers but also any developer that reviews code on a frequent basis could participate in this study.

5.2 User Study

The design for the study was as follows. First a pilot study was deployed, followed by the user study. We describe the methodology and possible threats to validity for these studies. Both had a similar setup and the feedback was gathered in an evaluation survey online.

5.2.1 Methodology

In this section we discuss our approach to evaluate the tool based on our hypotheses in the thesis statement in Section 2.3 and research questions stated in Section 5.1. The evaluation of CDV was performed in two steps. A pilot study was deployed, and the feedback was gathered in an evaluation survey. Participants where professional developers that used the developed tool for at least one merge request. Subsequently, with some of the improvements from the pilot study, a user study was deployed. The feedback was collected using a slightly adapted evaluation survey based on the pilot study participants feedback. Again, only professional developers that utilized the tool for at least one merge request where asked to participate in the study.

5.2.1.1 Pilot Study

The pilot study was deployed in the Swiss company *Business Systems Integrations AG* (BSI). BSI was founded in 1996 and counts over 320 employees meanwhile in 8 working places in Switzerand and Germany. Their core focus is software development for business applications, e.g., Eclipse Scout, BSI CRM[21] and BSI Studio[22] in combination with machine learning and cloud services. Participants were recruited through an internal board post describing briefly the problem stated in Section 2 and a preview of how we want to overcome this problem with the developed tool. The participants — all professional Java developers — had to utilize the tool CodeDiffVis for at least one code review. To test the tool, participants had two weeks of time because the number of assigned reviews may vary per developer. All participants received an installation guide and a brief tutorial of how the graph of CDV works and what features and interactions are possible. These instructions are available in Appendix B.1 and B.2. After testing the tool, an evaluation survey had to be filled in online by each participant. Their feedback was used to make improvements on the software for the user study.

The structure of the survey was the following:

1. Welcome page
2. Usability questions
3. Feature questions
4. Usefulness questions
5. Demographic questions
6. Final questions

On the welcome page participants were briefly introduced to our research problem and our approach to solve it using a visualization. They had to accept the data policy settings by the University of Zurich and by Swiss and European law. Filling in the survey was completely anonymous. To ensure the right to delete participants' answers on request, they were provided with a generated and unique participant ID. In the first part, the usability questions, we asked the system usability score (SUS) questions defined by Brooke and questions about whether they

[21] `https://www.bsi-software.com/ch-de/crm.html`
[22] `https://www.bsi-software.com/ch-de/produkte/bsi-studio.html`

understood the features and color coding of nodes [Brooke, 1996]. In the second part, the feature questions, questions targeted specific features, e.g., whether they think it would be useful to see if a node is public or private, or if the type of node (class, abstract class or interface) would be relevant. Finally in the third part, the usefulness questions, we asked about whether they were able to find more issues than usually and up to what size of merge request the tool would work well. Beller et al. assessed the problems that can be fixed with modern code reviews. They categorized change-types based on the issues that were fixed with those changes. The two main categories are issues that lead to *evolvability changes* (non-bugs) and issues that lead to *functional changes* (bugs) [Beller et al., 2014]. Beller et al.'s definition was introduced to the participants to differentiate the two categories. One question targeted whether they were able to find more issues for each of the two categories separately. All questions that where not open questions were answered with a five-point Likert scale. After each block with single choice Likert scale questions, participants could specify their answers or tell their opinion about that topic in an open-ended question optionally. We collected various demographic data such as nationality, gender, age, working experience and average weekly workload (programming and code review). In the final questions participants where asked, whether they want to share their anonymous data for a public dataset that can be used by other researchers.

Upon completion of the survey, participants where also asked to voluntarily participate in a semi-structured interview. The idea was to provide them a platform to express their opinion about the tool more freely as in the evaluation survey.

5.2.1.2 User Study

The user study was deployed in the same Swiss company. Participants received the same installation guide and tutorial as the ones in the pilot study. The goal was to use the improved software from the pilot study and shift the questions from a usability perspective more to target our research questions. Participants where recruited directly by mail or phone. We included a further section to the usefulness questions of the survey to increase the understanding of the benefits of the tool to support code review: e.g., whether they understood the code change quicker or if they were able to navigate more easily through the changes and orient themselves better in the merge requests using CodeDiffVis' graph. Based on the feedback from the pilot study, we also underlined that participants should only answer based on their experience with the visualization tool CDV but not if they had i.e., issues with the setup of CDP. The same questions as in the pilot study to collect the demographic data where used. The survey questions are available in Appendix D.1.

After participating in the user study, participants were also asked to take part in a semi-structured interview.

5.2.1.3 Semi-Structured Interviews

After the participants signed a consent form, available in Appendix C, the interviews where held and recorded via Skype. Participants where first asked to speak openly about their opinion and experience with the tool. Subsequently they were asked what they think about the concept with the visualization for code reviews, and specifically, the call and dependency graph.

5.2.2 Threats to Validity

In the following section, we explain possible threats to validity for the user study, for both internal threats in Section 5.2.2.1 and external threats in Section 5.2.2.2. Furthermore, we explain how we mitigate the known threats and justify our decisions.

5.2.2.1 Threats to Internal Validity

Maturation. For the pilot study, software bugs where fixed and some improvements based on the participants feedback where implemented during the study phase. Due to that, not all participants experienced the same software version. The fixes and improvements concerned both parts of the software, the back-end parser and the front-end visualization. In the user study, one minor fix was implemented because of stability issues to generate the Json files in the back-end. No improvements or new features were added for the user study. During the pilot study, we concentrated exclusively on fixes and improvements that where necessary to successfully test the software and to comprehend the introduced concept. In the user study, we concentrated only on crucial stability issues.

Participant selection. The participant selection differed comparing the pilot and the user study. In the pilot study, participants where recruited with a blog post on the company's (BSI) internal board. Only people who were actively looking for such posts could be contacted this way. An advantage of this was that only very experienced and interested developers volunteered. In the user study, participants where recruited by calling them directly by phone, asking to join the study. It is possible, that participants were more likely to join the study because of peer-pressure or goodwill. To mitigate that effect, we made clear on multiple occasions that the participation is voluntarily and that they can drop out of the study at any time without providing any reason.

Testing. If participants did not manage to set up CDP in a reasonable amount of time, they were allowed to utilize a sample merge request that was provided with the software. It is possible that these participants did not experience CDV the same way compared to those who used the software for professional merge requests. Only one participant used that option.

Attrition. None of the participants dropped out of the study. Besides, some participants needed a regularly reminder to test the software and to fill in the survey.

Instrumentation. In the user study, we shifted the survey's perspective more to match the research questions of this thesis. We also explicitly asked to only answer based on the experience with the visualization tool but not if participants had issues installing or setting up the software, especially the back-end part.

Social interaction. For the user study, participants where predominantly selected from the same working place in Zurich. It is possible that participants may have interchanged about the study which could have influenced the results in either positive or negative way.

5.2.2.2 Threats to External Validity

Sampling bias. All participants where employees of the Swiss company BSI. Code review is a required procedure in software quality evaluation at said company. Most developers showed at least 2 years of programming and code review experience. Furthermore, they solely use GitLab for the code reviews and no visualization tool. We believe, despite the narrow participant selection; the aforementioned factors would make the results applicable generally to developers.

History and aptitude-treatment. A few weeks before the studies started, due to SARS-CoV-2, Switzerland and Germany were put into *soft lock-down*, i.e. restrictions on outdoor activities, ban on public meetings and enforcement of home office, where possible. Participants may have experienced a higher stress level which could have influenced their answers in the evaluations. An advantage of this effect was that participants may have had more free time to test the tool, i.e., because they did not have to travel to the working place.

Hawthorne effect [McCarney et al., 2007]. We tried to mitigate this effect by enforcing participants to use their familiar working place and settings and that they used productive merge requests to test the tool. Only one participant used the provided sample merge request.

5.2.3 Results

In this section, we will outline the results from the user study. We start with the demographic questions and an analysis of the SUS. Thereafter we report important or surprising findings from the survey answers and from the semi-structured interviews.

5.2.3.1 Pilot Study

Four senior professional developers, all male, aged 30 – 35 years (32.75 years average), from the Swiss company BSI participated in this pilot study. Two participants where Swiss and the other two German. All showed at least 3 – 5 years of programming experience in a professional setting. Two of the participants claimed to have over 11 years of programming experience. The answers of all participants did not differ when asked explicitly about their experience with the programming language Java. All of the participants where experienced code reviewers with at least 3 – 5 years of code review experience. One participant claimed to have over 11 years of code review experience in a professional setting. Their average weekly workload spent on programming was 16 hours (8.98 hours standard deviation and 17.5 hours median) and their average weekly workload spent on code review totals up to 8.75 hours (5.5 hours standard deviation and 7.5 hours median). One participant volunteered to take part in a semi-structured interview.

The average SUS for the pilot study was 59.38 (standard deviation 28.69 and median 75). The statement *"I think I would need the support of a technical person to be able to use CDV"* scored the least (SUS of 25) since participants agreed the most. However, the statement *"I would imagine most people would learn to use CDV very quickly"* scored the best (SUS of 81.25) since participants agreed the most.

One of the primary outcomes of the pilot study's interview was the proposal of a reverse highlighting of nodes when hovering the code in the merge request. This feature was later implemented as the *reverse navigation* feature. Another outcome was to put the visualization in a separate window that can, i.e., be opened on another screen. Prior to that the visualization was integrated in the GitLab page and overlayed parts of the code in the review.

5.2.3.2 User Study

Nine professional developers, thereof seven male, 1 female, aged 26 – 47 years (33.50 years average), from the Swiss company BSI participated in this study. Eight participants were Swiss. One participant did not share age and gender nor nationality. The participant's experience in programming, Java and code review varies over all 9 participants. In Figure 7 an overview of participant's experience is provided. Their programming experience ranged from less than 1 year to 11 years or more. Most participants claimed to have 6 – 10 years of programming experience. The answers of all participants did not differ when asked explicitly about their experience with the programming language Java. Their code review experience ranged from less than 1 year to 11 years or more. Most participants claimed to have 2 years of code review experience. Their average weekly workload spent on programming was 28.2 hours (8.3 hours standard deviation and 30 hours median) and their average weekly workload spent on code review totals up to 5.4 hours (3.28 hours standard deviation and 4 hours median). Two participants volunteered to take part in a semi-structured interview.

The average SUS for the user study was 72.22 (standard deviation 20.34 and median 75). The statements *"I found CDV very cumbersome to use"* and *"I felt very confident using CDV"* scored the least (SUS of 66.67 each) since participants agreed in both cases the most. Though, the statement *"I thought there was too much inconsistency in CDV"* scored the best (SUS of 80.56) since participants agreed the least.

In Figure 8, participant's answers are summarized in five plots. Overall, there is a slight tendency towards a right shift in the data, meaning participants rather agreed with the statements.

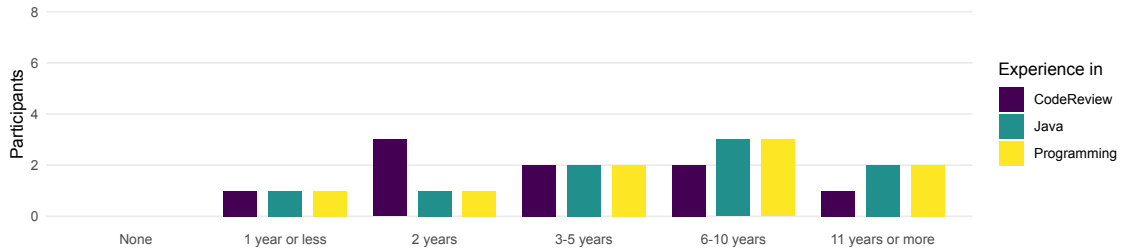
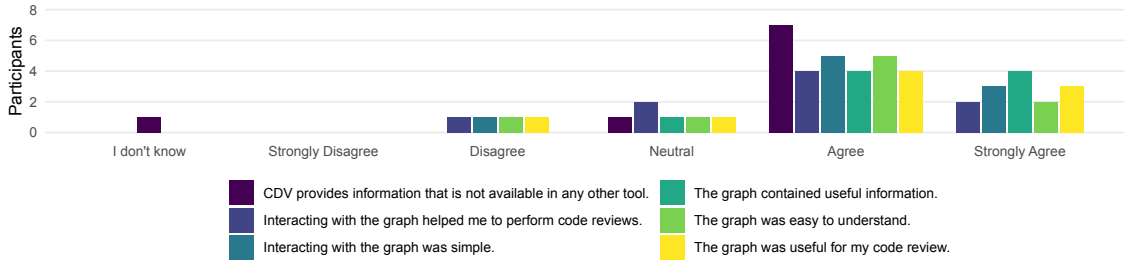


Figure 7: Participant's experience in programming, Java and CodeReview for all 9 participants of the user study.

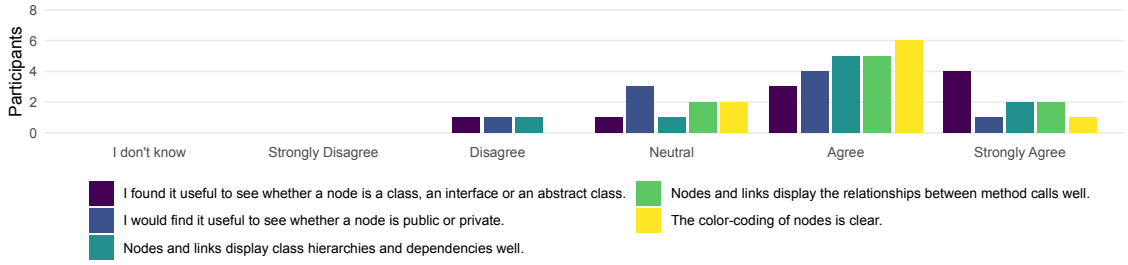
This is especially visible in Figure 8a and 8b. Most divergencies emerged with the questions about merge request size and the general benefits of the tool, as shown in Figure 8d and 8e. Some participants argued that *"for large code reviews, the graph is too dense – even for classes only"*, while others thought that *"for a small merge request the graph blew up so much that I had trouble finding the changed classes"*. Other participants where completely happy with the visualization, stating, i.e., *"I used it [CDV] in a huge merge request where it was very useful to span all dependencies"*. For Figure 8c a surprising result is that some participants mentioned that they where able to find more issues, for both evolvability and functional changes. None of the participants mentioned that CDV impeded them in detecting issues. One participant argued *"I would not say that CodeDiffVis lead to more [issue] detection per se but it lead 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 [issue] detection"*.

In general it seemed that participants either felt very excited about the tool and the concept behind it or they didn't like it at all. Therefore, their feedback was mixed. I.e., one participant concluded *"CodeDiffVis is a great way to get a better overview of the coherence of code changes and thus helps to improve the quality of code reviews and the quality of software in the long run. A cool approach, I would be happy to use it in the future"* while another participant said *"the two dimensional projection of code is not necessarily intuitive (e.g. like for road maps). The relations in the physical environment can't reflect the class network relations. Maybe the visual information confuses. I suppose a graph is not the optimal solution"*. The participant in the second interview summarized this misconception in the following way:

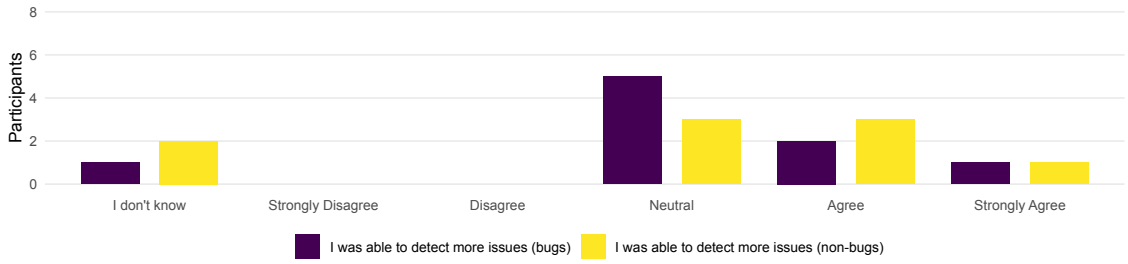
> *"Good tool, well implemented. It is a good tool to get an overview on a merge request. I liked the change-based coloring; it is very useful. But I thought I had "too much" freedom. I didn't really know what was right to do, where to start or how to use the tool properly."*
>
> (participant of interview 2)



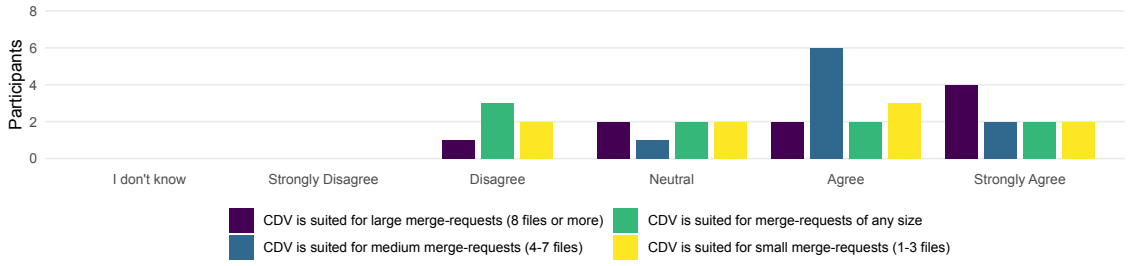
(a) Result of the general questions on CDV.



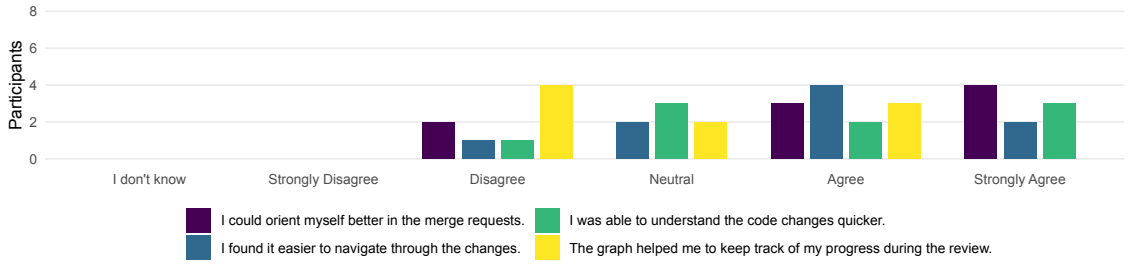
(b) Result of the questions on specific features of CDV.



(c) Result of the questions about the issue type detected.



(d) Result of the questions about the size of merge request that CDV would work best for.



(e) Result of the questions about the benefits of using CDV for code review.

Figure 8: Results of the user study (9 participants each question).

5.3 Online Survey

Different from the first two surveys, these participants did not get the chance to test the tool. To mitigate this problem, we added an introductory section explaining the tool and the features.

5.3.1 Methodology

The survey was distributed over various social media platforms: Twitter, Reddit and WhatsApp. We additionally used the retweet option and multiple subreddits to post a link to the survey. The structure of the survey was the following:

1. Welcome page
2. Questions on visualizations for code review
3. CodeDiffVis
4. Demographic questions
5. Final questions

After the welcome page, the survey started with general questions about whether the participants think code review would benefit from visualization tools and what kind of tools they know and or use, if any. In the second part, CDV was introduced with images, and the graph was explained using the same sample merge request from Eclipse Scout as in Figure 5. Subsequently, for each kind of the interaction described in Section 4.1.3, a YouTube video was embedded in the introductory section explaining the features with a captions comment. Participants where asked to rate the features based on whether they understood how it worked and if it would be useful for a code review using a five-point Likert scale. Further, we removed the questions about the type of issues found and the applicability to merge request size but we kept the questions that target the thesis statements. Furthermore, we included a demographic question about the job position of the participants and on what regular basis they do programming and code review. All questions of the online survey can be found in Appendix D.2.

5.3.2 Threats to Validity

In the following section we explain possible threats to validity to the online survey, for both internal threats in Section 5.3.2.1 and external threats in Section 5.3.2.2. Furthermore, we explain how we mitigate the known threats and justify our decisions.

5.3.2.1 Threats to Internal Validity

Maturation. One participant mentioned that the videos explaining the features of CDV where not fully conclusive. To mitigate this problem for further participants, captions were added to the videos explaining the tool. From participant ID 67, the videos were presented with captions.

Participant selection. The survey was distributed in the following way. First, developers that could not participate in the user study where directly contacted. Second, the survey was distributed and retweeted over Twitter. And third, on Reddit, the survey was distributed in different *Subreddits*, such as Code Review, Software Engineering and Computer Science. Only developers that are part of any of these communities could participate in the study.

Attrition. The majority of participants dropped out of the survey before completing it. Off 131 participants, 67 reached and answered the first part with questions about visualizations for code review. Participants that dropped out of the survey at this stage only looked at the general instruction page where no question was asked. 29 of the 67 participants reached and answered the second part where CDV was introduced and completed the survey.

5.3.2.2 Threats to External Validity

Situation effect. It is possible that the timings when the survey was posted on Twitter and Reddit where not optimal. In a sense that it would not be representative for a "global population" of developers because of the day and night cycle, posts disappear easily after a few hours or days. Furthermore, developers that do not use any of these media did not have a chance to participate.

5.3.3 Results

In this section, we will outline the results from the online survey. We start with the demographic questions and we point out important or surprising findings from the survey answers.

67 people, participated in this study. Thereof 29 participants completed the survey, 23 male, 1 female, aged 20 – 43 years (30.61 years average). Eight participants where Swiss, the others where from various other countries including Germany, France, Brazil, Italy and many more. Some participants did not share age, gender or nationality. The participant's experience in programming, Java and code review varies over all 29 participants. In Figure 9 an overview of participant's experience is provided. Two participants claimed to program about once a month, seven said they program about once a week and 18 participants told to program on a daily basis. Two participants stated that they do code review about once a month, eleven told that they do it about once a week and eleven participants specified that they do code review on a daily basis. Their programming experience ranged from less than 1 year to 11 years or more. Most participants claimed to have 3 – 5 years or 11 years or more of programming experience. The answers of all participants did differ greatly when asked specifically about their experience with the programming language Java. Most participants where not experienced Java programmers (1 year or less). Their code review experience ranged from less than 1 year to 11 years or more. Most participants claimed to have 2 years of code review experience. Their average weekly workload spent on programming was 18.82 hours (14.8 hours standard deviation and 20 hours median) and their average weekly workload spent on code review totals up to 5.63 hours (6.65 hours standard deviation and 3 hours median). Of the 29 participants that completed the study, 21 where professional developers, seven were academic researchers, three industrial researchers, four project managers, five spare-time developers and five students. Multiple answers where possible.

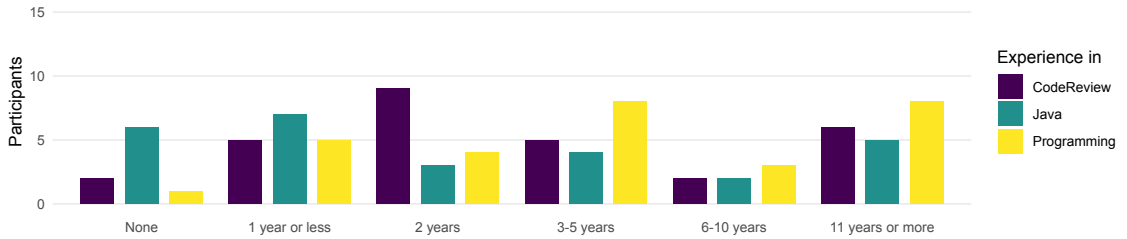


Figure 9: Participant's experience in programming, Java and CodeReview for all 29 participants of the online survey.

In Figure 10 participant's answers are summarized in five plots. Generally participants think that code review would benefit from visualization tools, although such tools seem to be unknown by the majority. Not all participants that know visualization tools for code review also use them (17 of 26 participants). Similar to the user study, there is an overall slight tendency towards a right shift in the data, where the responses concerning CDV are predominantly positive. Most participants quickly understood how the graph layout works and what information is displayed, as visible in Figure 10c. Most features were also quickly understood. In Figure 10d, it is clear that the graph customization feature was the most difficult to understand. On the benefits of CDV,

as summarized in Figure 10e, most participants agree on that it would be useful to navigate and orient oneself through the changes, and that a reviewer would be quicker in understanding a change. Participants thought CDV would suit the least to keep track of the progress in a review. Nevertheless, the majority of people were interested in using CDV for their daily business.

We further analyzed the data to identify patterns such as which group of developers, i.e., experienced versus inexperienced, preferred CDV the most. We run Chi-squares tests for each case and analyzed the correlations between the variables. As expected, participant's age and programming experience represent dependent variables (p-value 0.0388). We found no effect between the current workload and participant's experience, meaning that experienced programmers or reviewers do not necessarily work more or less.

First, we analyzed if there is a pattern of participants that want to try the tool. We found no effect for experience or programming workload, but for participants' age (p-value of 0.0448). Participant's age correlated slightly negatively with answering positively to the question (Pearson correlation of -0.2758), meaning that older reviewers may be less likely to want to try the tool. We also analyzed whether this may be due to the effect that these participants already use or know tools that fit better for their code reviews, but this could not be verified. Whether participants think visualization tools are generally useful for code reviews also seems to be dependent on age (p-value of 0.0448, Pearson correlation of -0.2758). Subsequently we analyzed the four questions about the benefits of CDV, as summarized below:

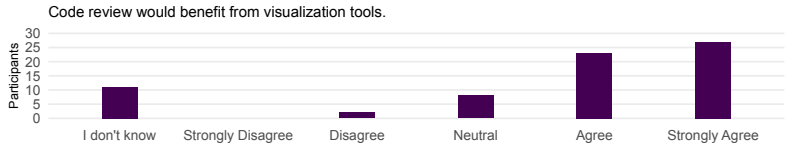
- *I would be able to understand code changes quicker.* An effect was found for participants age (p-value of 0.0184), i.e., participant's age and answer to this question are dependent variables. Agreeing with this statement correlated slightly negatively with participant's age (Pearson correlation of -0.2396), meaning that younger participants may be even more likely to agree with the statement.

- *CDV would help me to keep track of my progress during a review.* We found no effect in any of the observed variables. Either the question was generally difficult to answer or it is a matter of personal preference or caused by an unknown variable.

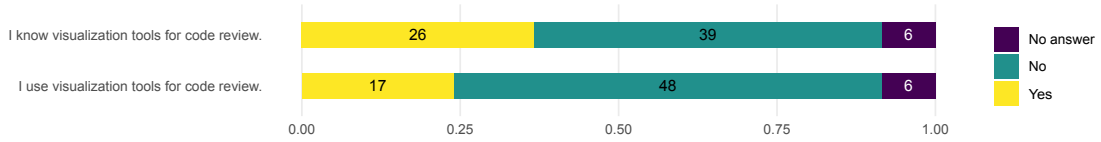
- *I could orient myself better in the merge requests.* We found an effect in participant's age (p-value of 0.0188, Pearson correlation of -0.3956).

- *I would find it easier to navigate through the changes.* We found a strong effect in participant's age (p-value of 0.0041, Pearson correlation of -0.3689).

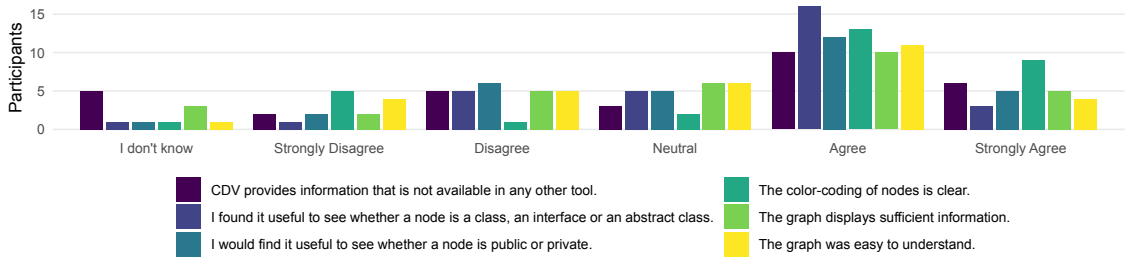
From the open questions after each block of Likert scale questions it becomes apparent that participants had a vastly different understanding on how visualizations could be helpful for code reviews. One participant thought: *"I think it could make it more interesting and people would like doing reviews a bit more"*. Another participant had concrete ideas what a visualization tool should do, such as: *"Code churn (#of times a line has been edited over time); "hot spots" of code change; flow charts of a message/object passing through the system"*. The participants conception of what a visualization should do seemed to reflect their opinion about CDV when answering. For instance, one participant that generally did not agree with the statements in the survey explained: *"the visualization should indicate which classes/methods went through impacting changes (a high churn) and which ones suffered less changes (low churn)"*. Few participants had trouble understanding all the features properly, especially the graph customization feature. Participants said that, e.g., *"it was not very clear how to delete a node"* or *"I did not quite understand the expansion of a node. The deletion was clear"*. Overall, participants seemed to like CDV, besides mentioning some finishing work that needs to be done, e.g.: *"CDV is clearly an improvement over the flat list in GitLab, but it needs more work"*, *[The] color-coding is not really colorblind-friendly"* or *"Please, keep improving your tool!"*.



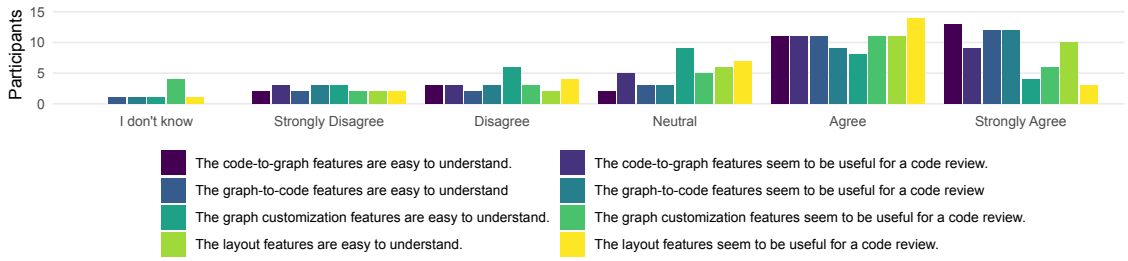
(a) Result of the question on the benefits of visualization tools answered by 67 participants.



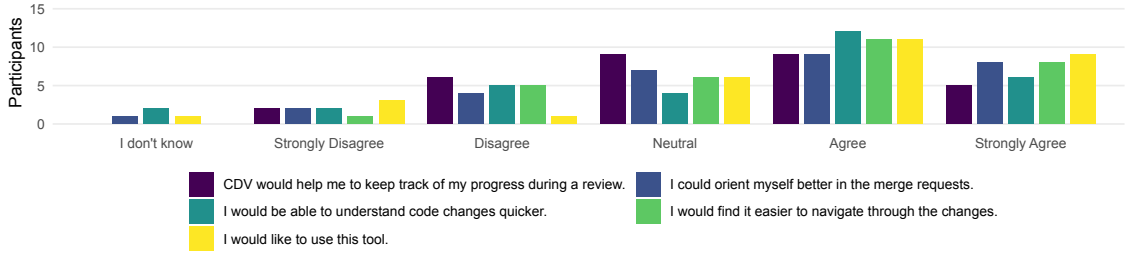
(b) Result of the question on whether participants know and / or use any visualization tools for their code reviews, answered by 67 participants.



(c) Results on the introductory questions on CDV, answered by 29 participants.



(d) Result on the questions on specific features, introduced with images and videos, answered by 29 participants.



(e) Results on the questions on the benefits of CDV, answered by 29 participants.

Figure 10: Results of the online survey (67 participants, respectively 29, each question).

6 Discussion

To discuss the results, we comment each of the research questions from Section 5.1 and the thesis statement from Section 2.3 separately.

RQ1 *What is a reviewer's perception of CodeDiffVis as a means to support code review in a productive working environment?* The majority of professional developers expressed themselves positively regarding CDV. An SUS of 72.22 is considerably good [Bangor et al., 2009]. We believe this is an excellent result for a prototype (CDV) that we used to evaluate the concept of a graph as a visual support for code review. Clearly there is room for improvements from a usability perspective, for example, the arrangement of nodes and the spaces between nodes and the flickering of nodes when hovering over them quickly could be improved in a later version. Participants predominantly agreed on the statements that CDV would make it easier to navigate through the changes, or that they could orient themselves better in the merge request. There was a mixed feedback whether the graph actually helps to understand the code changes quicker because most participants where unsure about that statement (neutral answer). The statement about whether CDV helps to keep track of a review was agreed on the least. We speculate this is due to what participant of interview 2 mentioned about having *"too much freedom"* with CDV and that there is *"no correct way to use it"*. Based on the aforementioned reasons and evaluation, we believe that the participants perception of CDV as a means to support code review is generally positive and helpful.

RQ2 *What is a reviewer's perception of CodeDiffVis as a means to support code review when evaluated as a concept?* It was a challenge to design a survey in a way that a participant gets enough information about the concept and the developed tool without actually providing the tool to the participants. We believe the approach with videos and images worked well since, based on the participants feedback, there were only few issues with understanding the explanations. Important to notice is that almost all participants thought that a visualization could improve code review but only a fraction of participants (about 25%) actually claimed to use such tools. One participant of the online survey argued that *"too often too many [visualization] tools [for code review] face their ends too soon [before accomplishing anything]"* and the participant of the second interview in the user study mentioned *"most of the graphical tools are just games"*. It was not very surprising to find that generally younger developers thought CDV was more helpful than older reviewers. This could just be due to the fact that older developers may have their fixed set of tools and would find it harder to switch to new things. Although for the statement about whether CDV helps to orient oneself better in the merge request, the aforementioned finding could not be verified. This could be due to how the reviewers work, for some a graph might be helpful while for others it is just a distraction. Overall we believe the participant's perception of CDV was mostly positive i.e. that it would be helpful for code reviews.

Thesis Statment *Visualization techniques support a reviewer during a code review. A useful visualization would be one that displays method calls and dependencies among classes that are part of the change.* According to the participants answers from the online survey, visualization tools would be beneficial for code reviews. We did not evaluate this statement in a quantitative study but in a qualitative study, based on the participants opinions. From both studies, it is not clear whether CDV is considered helpful because it uses the graph visualization or just because there is supplementary information available for a merge request. On contrary, it would be challenging to visualize dependencies among classes and method calls without using a graph-like layout. We summarize the results regarding the thesis statement as follows. Most participants thought CDV would help them to orient themselves better in a merge request and navigate easier through the changes. Many participants also thought CDV would even help them understanding a code change quicker. Finally, the approach of visualizing method calls and dependencies among classes explores the right direction since most reviewers seem to miss this information in their review tool of preference.

7 Conclusion

In this final section we conclude the work and outline some future work in Section 7.1 that addresses unanswered questions or questions that newly emerged with the results of this thesis from a research perspective, or to polish the developed tool CDV for productive use.

Code review is an essential procedure in modern software development. We developed a tool, CodeDiffVis, that supports a developer during code reviews. We used a graph visualization to display class dependencies and method calls among the changes. The interactive visualization integrates into the web-based code review tool GitLab. We evaluated the usefulness of the tool in two qualitative studies. First, a user study with professional developers in the Swiss company BSI was deployed to evaluate the visualization concept and generally the tool in practice. A preceding pilot study deployed in the same company was used to improve the software beforehand. And Second, an online survey was deployed and distributed over social media platforms to gather a broader feedback on the concept of the tool. These participants where introduced to CDV with explaining images and videos. In both studies, participants responded rather positive about the proposed solution. We observed a slight tendency, that younger programmers and code reviewers where even more likely to find the tool useful. With these results, new questions arise such as *how* do developers use the tool in practice or what about other features such as supporting a broader variety of programming languages. Such questions are addressed in Section 7.1.

7.1 Future Work

The aspects to polish the tool for productive use and to address future research questions can not be fully separated. In the following final section, we provide a non-conclusive list of possible improvements and aspects that might want to be considered in future.

1. *Code Change Ordering.* By implementing the theory of code change ordering from Baum et al., i.e., as a *guided tour* through the merge request, a reviewer could more easily keep track of what classes where already reviewed [Baum et al., 2017].

2. *Usage Statistics and Metrics.* We only evaluated the tool based on the participant's opinions in the surveys. An improvement would be to study how the developers use the tool in practice and what features they use the most. By collecting usage data automatically, a future study could concentrate on this quantitative analysis.

3. *Graph Layout.* The current graph layout is not optimal because related nodes are not necessarily drawn close to each other and links often overlap.

4. *Referenced Code Declarations.* Some participants expected the feature of being able to also jump to referenced nodes, i.e., nodes that are not part of the change. This would, i.e., prevent reviewers having to switch to their IDE during a review.

5. *Change Impact and Usage.* A often mentioned missing feature was to show the change impact. Also incoming references could be addressed, i.e., how many methods, not part of the code change call a changed method that is part of the change.

6. *Supported Programming Languages, Browsers and Frameworks.* CDV and CDP only support Java, Google Chrome and GitLab. First, programming languages such as Python or JavaScript could be added to achieve a broader user acceptance. Second, CDV could be extended to work with other web-browsers like Firefox or Safari. And third, other code review platforms that function in a similar way could be integrated, i.e., GitHub or BitBucket.

References

M. Aeschlimann, D. Bäumer, and J. Lanneluc. Java tool smithing extending the eclipse java development tools. *Proc. 2nd EclipseCon*, 2005.

A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

A. Bangor, P. Kortum, and J. Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

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

T. Baum, O. Liskin, K. Niklas, and K. Schneider. A faceted classification scheme for change-based industrial code review processes. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 74–85. IEEE, 2016.

T. Baum, K. Schneider, and A. Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–340. IEEE, 2017.

M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211, 2014.

A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464, 2010a.

A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512, 2010b.

J. Brooke. Sus: A "quick and dirty" usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, editors, *Usability Evaluation in Industry*, pages 189–194. Taylor and Francis, London, 1996.

M. Carraz, K. Korakitis, P. Crocker, R. Muir, and C. Voskoglou. State of the developer nation 18th edition - q4 2019: The latest trends from a survey of 17,000+ developers, 2019. `https://www.developereconomics.com/resources/reports/state-of-the-developer-nation-q4-2019`.

K. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2011.

R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073. IEEE, 2012.

M. Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

E. Fregnan, T. Baum, F. Palomba, and A. Bacchelli. A survey on software coupling relations and tools. *Information and Software Technology*, 107:159–178, 2019.

L. Fuentes-Fernández and A. Vallecillo-Moreno. An introduction to uml profiles. *UML and Model Engineering*, 2:6–13, 2004.

T. Ho-Quang. *Empowering Empirical Research in Software Design: Construction and Studies on a Large-Scale Corpus of UML Models*. PhD thesis, Chalmers University of Technology and University of Gothenburg, 2019.

T. Ho-Quang, M. R. Chaudron, G. Robles, and G. B. Herwanto. Towards an infrastructure for empirical research into software architecture: challenges and directions. In *2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 34–41. IEEE, 2019.

J. Jenkov. Understanding dependencies. `http://tutorials.jenkov.com/ood/understanding-dependencies.html`, 5 2014. Accessed: 2020-06-30.

O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038. IEEE, 2016.

R. McCarney, J. Warner, S. Iliffe, R. Van Haselen, M. Griffin, and P. Fisher. The hawthorne effect: a randomised, controlled trial. *BMC medical research methodology*, 7(1):30, 2007.

S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

M. D. Plumlee and C. Ware. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(2):179–209, 2006.

P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. Open source peer review–lessons and recommendations for closed source. *IEEE software*, 29(6):56–61, 2012.

B. Tate and J. Gehtland. *Better, faster, lighter java*. "O'Reilly Media, Inc.", 2004.

D. van Bruggen, S. Viswanadha, and J. V. Gesser. Javaparser. `http://javaparser.org`, 2008. Accessed: 2020-06-30.

R. J. Wirfs-Brock. Characterizing classes. *IEEE software*, 23(2):9–11, 2006.

Y. Yu, H. Wang, G. Yin, and T. Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74: 204–218, 2016.

A Design Mockups

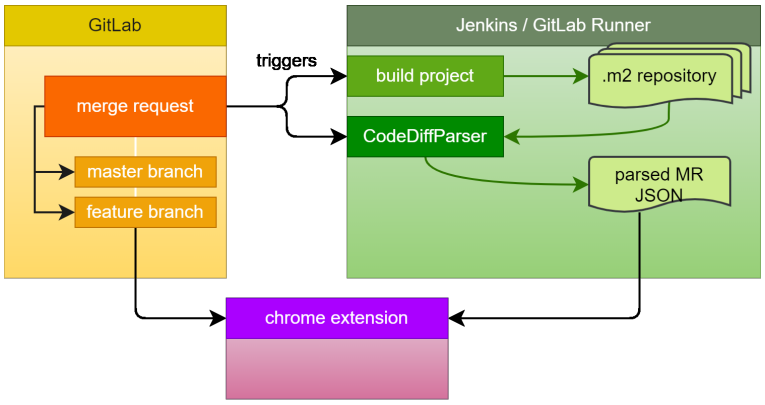


Figure 11: Inital plan of the software architecture to automize MR analysis with the tool. CodeDiffParser would be put on a Jenkins server and CodeDiffVis is the Google Chrome extension.

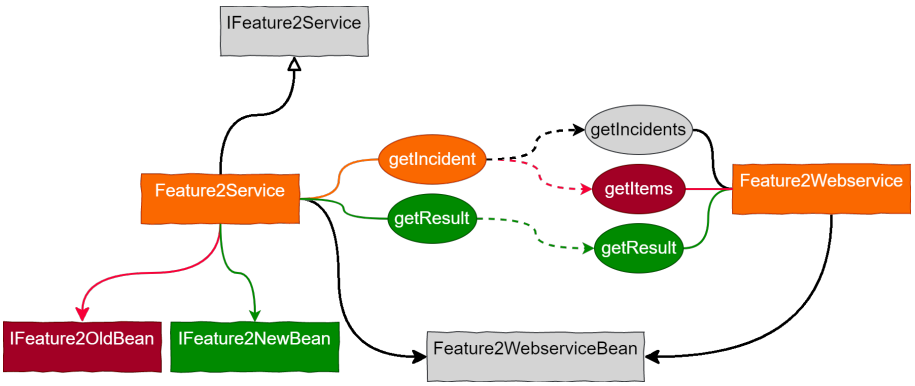


Figure 12: Example mockup of a class view and changed, added and deleted parts.

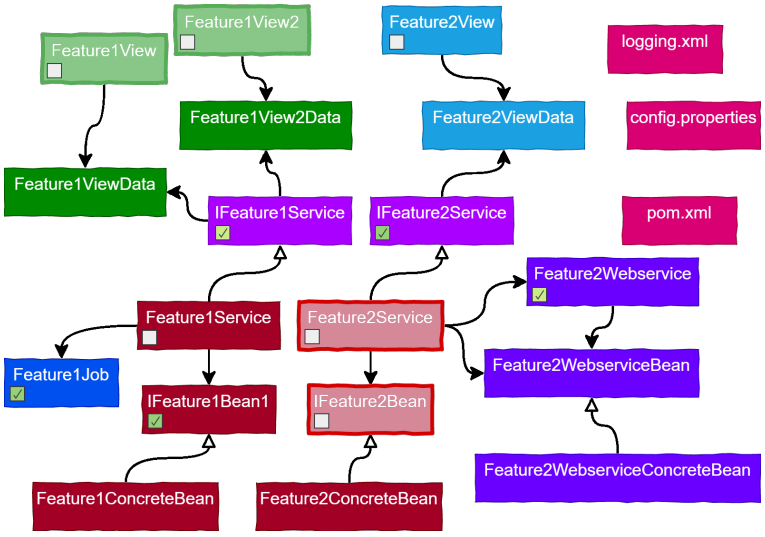


Figure 13: Example mockup of a merge request view. Already reviewed nodes have a checked checkbox, unread nodes are drawn with a thick border.

B Installation and Tutorial for the User Study

B.1 Installation Guide

Code-Diff Parser and Visualization

Thank you for participating in this study. This tutorial will guide you through the installation process of the software. It should take about 15 minutes.

Requirements: Google Chrome, Java JRE 11 (or higher), Git Console

1 Installation

Download the repository:

- **CodeDiffParser**: Java program, that analyzes the two Git branches per merge request
- **CodeDiffVis**: Chrome extension for the visualization

1.1 CodeDiffParser

You need to execute the following for each merge request manually. Later, the idea would be to install the parser on a Jenkins such that it automates that process.

You can use the pre-compiled version of CodeDiffParser or you can compile it yourself. Copy JAR and the script `run_codediffparser.sh`, i.e., to `C:\dev\study`.

Use a Unix-like command line tool (e.g. MINGW64) and execute the script `run_codediffparser.sh`. The JAR of the parser needs to be in the same directory.

- `./run_codediffparser.sh source-branch target-branch git-project-url local-maven-repo`
  - `./run_codediffparser.sh myfeature master https://gitlab.com/group/project.git ~/.m2/`

> After a successful execution, copy the resulting out.json into your CodeDiffVis directory. Chrome does not allow access to any other directory except to the extension itself.

1.2 CodeDiffVis

1. In Google Chrome, under chrome://extensions/ enable the «developer mode» (top right)
2. Choose «load unpacked» and select your CodeDiffVis directory (e.g. `C:\dev\study\CodeDiffVis`)
3. On the top right of your browser you should see a greyed-out symbol of CodeDiffVis

If you have already installed a previous version of CodeDiffVis, to update the extension you must click its reload button in the extension overview.

Now you are ready to proceed with «Tutorial – CodeDiffVis».

B.2 CodeDiffVis Tutorial

CodeDiffVis

Thank you for agreeing to participate in this study. This guide briefly explains the features of the software.

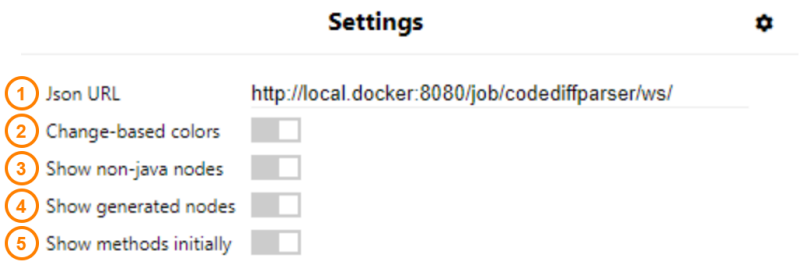
CodeDiffVis uses a so called ContentScript (JavaScript), which will be injected into the webpages. Upon calling a merge request site, the script is activated. The extension uses the «local storage» of Chrome and saves your settings.

If you open a merge request site on GitLab, a separate window will open where the graph will be drawn. The graph is only interactive for as long as you have opened the merge request tab in the browser.

For very big merge requests (>20 files) it is possible that loading the graph takes some time. This is due to the fact, that GitLab is highly resource demanding.

1 SETTINGS

If you are on a merge request site, on «/diff», click on the symbol of CodeDiffVis.

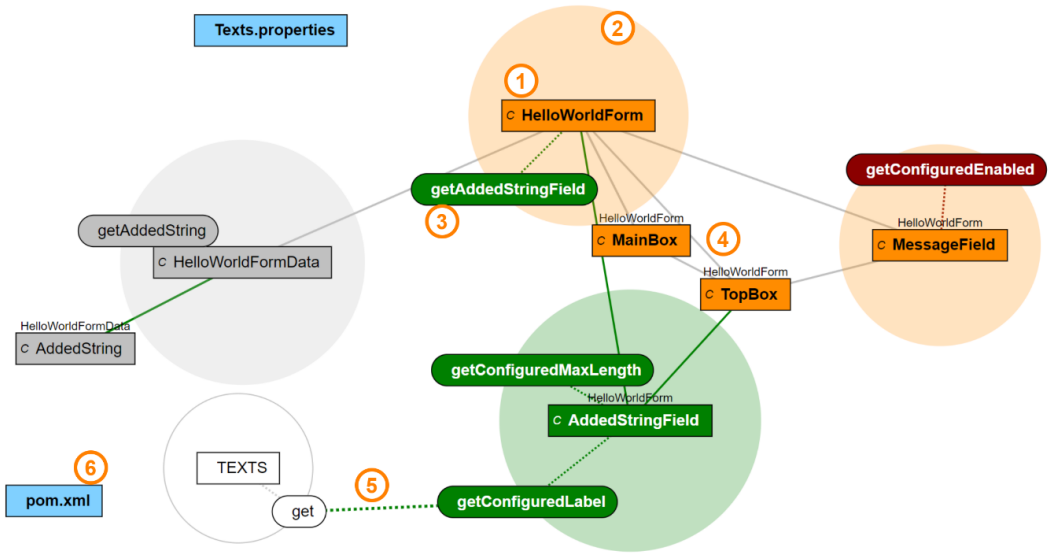


1. Path to Json file. It i seither relative tot he location of CodeDiffVis on your computer or absolute via web url. If no name is specified, the merge request ID is used (e.g. 1234.json). Two examples:
   - out.json
   - http://local.docker:8080/job/codediffparser/ws/
2. Toggle «change-based colors» colors the nodes based on whether they are different in both branches. Disabling this toggle will allow coloring by package. For change-based coloring, the following schema is used:
   - *Ret*: deleted, i.e., this node is only present in the target branch
   - *Green*: new, i.e., this node is only present in the source branch
   - *Orange*: changed, i.e., within this node, there were changes (not counting comments)
   - *Grey*: generated nodes that have the `@Generated` annotation
   - *White*: referenced nodes, i.e., not part of the change
   - *Light blue*: «non-Java nodes », i.e., JavaScript, Properties, XML's etc.
3. Toggle, if all nodes should be drawn or only Java nodes
4. Nodes that have the `@Generated` annotation can be hidden
5. Decides whether methods should be added initially

2 THE GRAPH – BRIEFLY EXPLAINED

The graph is a mix between a «call graph» and a «dependency graph». On the one hand, method calls are drawn and on the other hand dependencies, e.g., class hierarchies, imports or interface references are shown.

We recommend opening the graph on a separate screen and to use the full screen mode (F11).



1. Represents a Java class. Classes are denoted with C, Abstract classes with A and Interfaces with I
2. The circle is added to nodes that contain at least one method in the code change
3. Rounded corners of a node indicate a method
4. Inner classes are denoted with a label on the top with the corresponding class file name
5. Represents a method call. "getConfiguredLabel" of class "AddedStringField" calls "get" from "TEXTS"
6. Nodes that do not represent Java components are shown with their full name

3 INTERACTIONS

The embedded graph is not static. You have a lot of options to configure it to your needs such that it becomes useful for your code review.

For example, you can remove already reviewed nodes from the graph, or you can show only parts of the graph with the hovering function.

Name	*Aktion*	*Beschreibung*
Pan	Move the graph	Hold and click (left) on the free space and move the mouse to move the graph
Drag	Move nodes	Change positions of nodes. Drag a node to your desired location
Zoom	Zooming	Use the mouse wheel to zoom in and out
SVG resize	Enlarge / shrink the view	You can resize the window arbitrary
Hover	Highlight connections	Hover over a node with the mouse; connected nodes will be highlighted
Hover reverse	Highlight nodes	Hover over the source code in the merge request; the node and all connected nodes are highlighted in the graph. The graph automatically centers your node
Hover lock	Pause highlight	With **CTRL** you can lock / unlock the current highlight to prevent flickering
Link	Jump to declaration	On **click** (left) on a node, jump to the first occasion in the code change directly. A node remembers being clicked and changes its appearance to make it easier for you to track which nodes are already reviewed
Expand	Add referenced nodes	With **SHIFT + click** (left) referenced nodes can be added
Remove	Remove nodes	Remove any node with a **click** (right)

C Interview Consent Form

Zurich Empirical Software Engineering Team (ZEST) University of Zurich
Department of Informatics
Binzmühlestrasse 14
CH-8050 Zurich
Switzerland

Contact Person
Josua Fröhlich
josua.froehlich@uzh.ch

Interview - Participant Consent Form

You are invited to participate in a research interview that is being conducted by Josua Fröhlich.

Josua Fröhlich is writing his master thesis in the ZEST at the Department of Informatics, University of Zurich. This work is supervised by Prof. Dr. Alberto Bacchelli, who can be contacted at BACCHELLI@IFI.UZH.CH.

Purpose

The purpose of this research interview is to collect feedback based on the participants' experience with the developed tool (CodeDiffVis).

Study Procedure and Collected Data

If you consent to voluntarily participate in this research interview, your participation will include recording the interiew.

Potential Risks

In case you perceive your participation in this study as stressful, you have the right to terminate your participation or to not answer a question. In case of further questions or doubts, you may contact any of the researchers (contacts are provided above) or the OEC Human Subjects Committee of the University of Zurich at HUMAN.SUBJECTS@OEC.UZH.CH.

Benefits

By participating in this study, you will receive a symbolic reward in form of a beer or an alcohol-free beverage.

Conditions of the Study

- **Voluntary Participation.** Your participation in this research must be completely voluntary. Even if you do decide to participate, you can withdraw at any time without any explanation. If you do withdraw from the study, your data will be erased and will not be included in any analysis.
- **Possibility not to answer or withdraw details from your interview.** You are free to not answer all of the questions we ask you. Also, at any moment, you can ask us to remove a specific information from your answers.
- **Confidentiality.** The research staff will protect your personal information closely so no one will be able to connect your responses and any other information that identifies you. Official investigations may require us to show information to university or government officials (or sponsors), who are responsible for monitoring the safety and procedure of our research. Directly identifying information (e.g., names, addresses) will be kept strictly separate at all times from any other collected data and will be stored in a different location. Furthermore, it will not be associated with

the data after it has been analyzed. In particular, your contacts, this approved consent, and the recordings of your interview will be securely stored at the University of Zurich in an encrypted archive that only the research staff of ZEST can open. You will not be identified in any publication that derives information from your research interview.

- **Dissemination of Data and Results.** It is anticipated that the analysis of this research interview will lead to results that will be shared publicly (e.g., in form of an academic publication in research conferences or journals). To allow for the reproducibility and reuse of our research, we ask your permission to also share an electronic version of (parts of) your transcribed interview data with the broad scientific community. Any personal information that could identify you will be removed or changed before files are shared with other researchers or results are made public. Data presented in presentations or publications will never allow identifying individual persons.
- **Your personal information.** In case you would be interested and willing to participate in future studies, we will put record of your contacts in a database (different than the one used for the interview data) securely stored at the University of Zurich in an encrypted archive that only the research staff of ZEST can open.
- **Possibility to withdraw data.** If, at any point, you decide that you would like your data to be completely withdrawn from the study, you are free to do so.

Contact for Information about the Study

If you have any questions or desire further information with respect to the study, you may contact Josua Fröhlich (JOSUA.FROEHLICH@UZH.CH) or Prof. Dr. Alberto Bacchelli (BACCHELLI@IFI.UZH.CH).

Consent for Study Participation

Your participation in this study is entirely voluntary. You are free to withdraw your participation at any point during the study, without giving any reason and without any negative consequence. Any information you contribute up to your withdrawal will be retained and used in this study, unless you request otherwise.

With your signature on this form you confirm the following statements:

- I understand the goal and procedures of the study and the applicable conditions.
- I had the opportunity to ask questions. I understood the answers and accept them.
- I am at least 18 years old.
- I had enough time to make the decision to participate and I agree to the participation.

In no way does this waive your legal rights or release the investigators or involved institutions from their legal or professional responsibilities.

Additional consent

[ ] I consent to publicly share my anonymized data
[ ] I would like to be contacted for future studies
[ ] I would like to receive the final publications based on my interview data

A copy of this consent will be left with you, and a copy will be taken by the researcher.

Name of Participant:

Location, Date:

Signature of Participant:

D Questionnaires

D.1 User Study Questionnaire

Code Review - Pilot Study

« Code review is a part of the software development cycle to improve code quality, i.e. the detection of bugs, knowledge transfer and team awareness. Code review can easily become a complex task. We created a visualization tool called *CodeDiffVis* to support code review. This work is part of my master thesis. I am studying Informatics with major in people-oriented computing and minor in bioinformatics at the University of Zurich. »

CodeDiffVis Survey

We designed this survey to assess the usefulness of CodeDiffVis during code review. This survey takes approximately 15 minutes to complete.

The questions in this survey target solely your experience with the visualization tool CodeDiffVis. If you had issues with the back-end, CodeDiffParser, or with installing the software, please refer to it in the very last question of this survey only.

The survey questions are in English only, yet feel free to answer in English, German, French or Italian.

There are 25 questions in this survey.

Participant ID

If you keep a copy of the following unique code, you can use it to request the removal of your responses from this survey at any time.

Please write your answer here:

Open Questions

Consider your experience with CodeDiffVis, what did you like most about it?

Please write your answer here:

What did you like the least?

Please write your answer here:

For which code changes - if any - did CodeDiffVis help you the most during the reviews?

Please write your answer here:

For which the least?

Please write your answer here:

Which version of CodeDiffVis where you using? *

❗ Choose one of the following answers
Please choose **only one** of the following:

- ◯ v1.1
- ◯ v1.0
- ◯ v0.4
- ◯ v0.3
- ◯ v0.2
- ◯ v0.1

Usability

State your agreement with the following statements on *CodeDiffVis* (CDV). *

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
I think that I would like to use CDV frequently.	◯	◯	◯	◯	◯	◯
I found CDV unnecessarily complex.	◯	◯	◯	◯	◯	◯
I thought CDV was easy to use.	◯	◯	◯	◯	◯	◯
I think that I would need the support of a technical person to be able to use CDV.	◯	◯	◯	◯	◯	◯
I found the various functions in CDV were well integrated.	◯	◯	◯	◯	◯	◯
I thought there was too much inconsistency in CDV.	◯	◯	◯	◯	◯	◯
I would imagine that most people would learn to use CDV very quickly.	◯	◯	◯	◯	◯	◯
I found CDV very cumbersome to use.	◯	◯	◯	◯	◯	◯
I felt very confident using CDV.	◯	◯	◯	◯	◯	◯
I needed to learn a lot of things before I could get going with CDV.	◯	◯	◯	◯	◯	◯

Visualization and Features

Please, evaluate the following claims on the *graph* of CodeDiffVis.

- With *graph* we always refer to the visualization of the dependency and call graph.
- With *interactions* we refer to all kind of possible interactions:
  - panning, dragging and zooming
  - clicking on nodes and navigation in the code change
  - adding and deleting nodes

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The graph was clear and easy to understand.	◯	◯	◯	◯	◯	◯
The graph was useful for my code review.	◯	◯	◯	◯	◯	◯
The graph contained useful information.	◯	◯	◯	◯	◯	◯
Interacting with the graph was simple.	◯	◯	◯	◯	◯	◯
Interacting with the graph helped me to perform code reviews.	◯	◯	◯	◯	◯	◯
CodeDiffVis provides information that is not available in any other tool.	◯	◯	◯	◯	◯	◯

Do you have any additional remarks about the *graph*?

Please write your answer here:

Please answer the following questions about the information displayed with the graph. *

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The graph displays sufficient information.	◯	◯	◯	◯	◯	◯
In the graph, I found it useful to see whether a node is a class, an interface or an abstract class.	◯	◯	◯	◯	◯	◯
In the graph, I would find it useful to see whether a node is public or private.	◯	◯	◯	◯	◯	◯
The color-coding of nodes is clear.	◯	◯	◯	◯	◯	◯
The nodes and links display the relationships between method calls well.	◯	◯	◯	◯	◯	◯
The nodes and links display class hierarchies and dependencies well.	◯	◯	◯	◯	◯	◯

Do you have any additional remarks about the information displayed with the graph?

Please write your answer here:

Applicability and Benefits

Please, evaluate the following claims on the issues found in code reviews using CodeDiffVis.

We categorize the following two code review related changes:

1. *Evolvability changes.* They refer to non-functional code changes for maintainability, not considered as bugs, i.e.:
   - renaming of variables or methods
   - missing documentation
   - extracting a method or interface into another/separate class
2. *Functional changes.* Changes in the programs' behavior, usually bugs, i.e.:
   - change of method logic
   - use of different resources
   - changes that led to unexpected or erroneous behaviour

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
I was able to detect more issues (non-bugs) that led to evolvability changes.	◯	◯	◯	◯	◯	◯
I was able to detect more issues (bugs) that led to functional changes.	◯	◯	◯	◯	◯	◯

Do you have any additional remarks about the issues found in code reviews using CodeDiffVis?

Please write your answer here:

Please, evaluate the following claims on CodeDiffVis (CDV).

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
CDV is suited for small merge-requests (1-3 files)	◯	◯	◯	◯	◯	◯
CDV is suited for medium merge-requests (4-7 files)	◯	◯	◯	◯	◯	◯
CDV is suited for large merge-requests (8 files or more)	◯	◯	◯	◯	◯	◯
CDV is suited for merge-requests of any size	◯	◯	◯	◯	◯	◯

Do you have any additional remarks about the applicability of CodeDiffVis in different code review sizes?

Please write your answer here:

Please, evaluate the following claims on the benefits of CodeDiffVis. *

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
With the graph, I was able to understand the code changes quicker.	◯	◯	◯	◯	◯	◯
The graph helped me to keep track of my progress during the review.	◯	◯	◯	◯	◯	◯
With the graph, I could orient myself better in the merge requests.	◯	◯	◯	◯	◯	◯
With the graph, I found it easier to navigate through the changes.	◯	◯	◯	◯	◯	◯

Considering the statements above, do you have any additional remarks about CodeDiffVis?

Please write your answer here:

Final Demographic Questions

What is your nationality?

Please write your answer here:

What is your gender?

Please choose **only one** of the following:

◯ Male
◯ Female
◯ Prefer to not disclose
◯ Prefer to self-define

How old are you?

❗ Only an integer value may be entered in this field.
Please write your answer here:

years

Your working experience as a developer in a professional setting; e.g. as an employee of an IT-company or as an Open-Source contributor.

Please choose the appropriate response for each item:

	None	1 year or less	2 years	3-5 years	6-10 years	11 years or more
For how many years have you developed software in a professional setting?	◯	◯	◯	◯	◯	◯
For how many years have you developed Java software in a professional setting?	◯	◯	◯	◯	◯	◯
For how many years have you performed code review in a professional setting?	◯	◯	◯	◯	◯	◯

How many hours per week, on average, do you spend on programming?

❗ Your answer must be at least 0
Please write your answer here:

hours

How many hours per week, on average, do you spend on reviewing code of other developers?

❗ Your answer must be at least 0
Please write your answer here:

hours

General Feedback

Thank you for filling in all answers so far. You are almost done.

In addition to analyzing your data in an anonymous form, may we also share your answers in a publicly available dataset?

(Selecting *yes* allows other researchers and the public to benefit from your answers and effort)

*

Please choose **only one** of the following:

◯ Yes
◯ No

Do you have any final remarks about CodeDiffVis or this survey?

Please write your answer here:

Thank you very much for participating.

You have already helped us a lot and we are especially grateful that you sacrifice some of you free time for our research.

In case you are interested, we are also searching for voluntaries to participate in a online interview (i.e. Skype). Depending on your preference and availability it takes approximately 30 minutes where you can express your thoughts on the tool more freely.

If you wish to participate, please write an e-mail to Josua Fröhlich (mailto:josua.froehlich@uzh.ch) so we can find a slot that fits for you.

04.06.2020 – 14:14

Submit your survey.
Thank you for completing this survey.

D.2 Online Survey Questionnaire

Study on Code Review - CodeDiffVis

Code review is a part of the software development cycle to improve code quality, i.e. the detection of bugs, knowledge transfer and team awareness. Code review can easily become a complex task.

In an attempt to support reviewers, we created a visualization tool called *CodeDiffVis* to support code review.

This work is part of my master thesis. I am studying Informatics with major in people-oriented computing and minor in bioinformatics at the University of Zurich.

CodeDiffVis Survey

We designed this survey to assess the usefulness of CodeDiffVis during code review. This survey takes approximately 15 minutes to complete.

The survey questions are in English only, yet feel free to answer in English, German, French or Italian.

There are 33 questions in this survey.

Participant ID

If you keep a copy of the following unique code, you can use it to request the removal of your responses from this survey at any time.

Please write your answer here:

Code Review Visualizations

According to the Cambridge dictionary, **visualization (https://dictionary.cambridge.org/dictionary/english/visualization)** is "the act or an example of creating an image, etc. to represent something".

Visualization tools automate this process by taking input data and generate the output accordingly: e.g., showing dependencies between objects as a graph.

Please rate the following statement.

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
Code review would benefit from visualization tools.	◯	◯	◯	◯	◯	◯

Why do you think a visualization tool would be helpful for a code review? *(Optional)*

Please write your answer here:

Why do you think a visualization tool would not be useful for a code review? *(Optional)*

Please write your answer here:

Please answer the following questions: *

Please choose the appropriate response for each item:

	Yes	No	No answer
Do you know any visualization tools for code review?	◯	◯	◯
Do you use any visualization tools for code review?	◯	◯	◯

What tools do you know? *(Optional)*

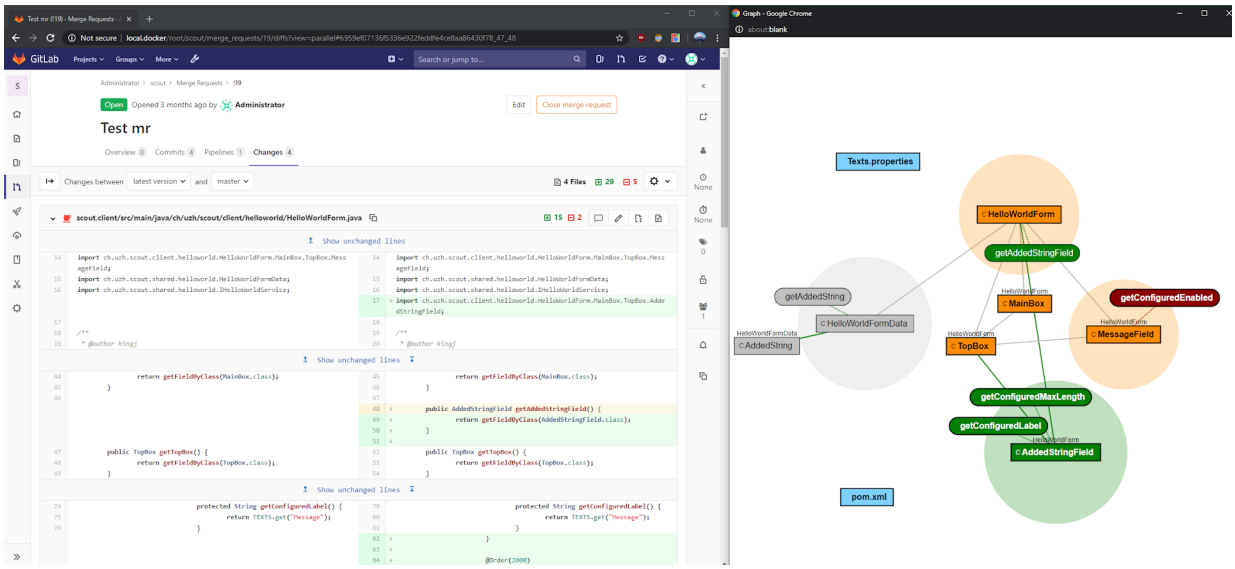
Please write your answer here:

What tools do you use? *(Optional)*

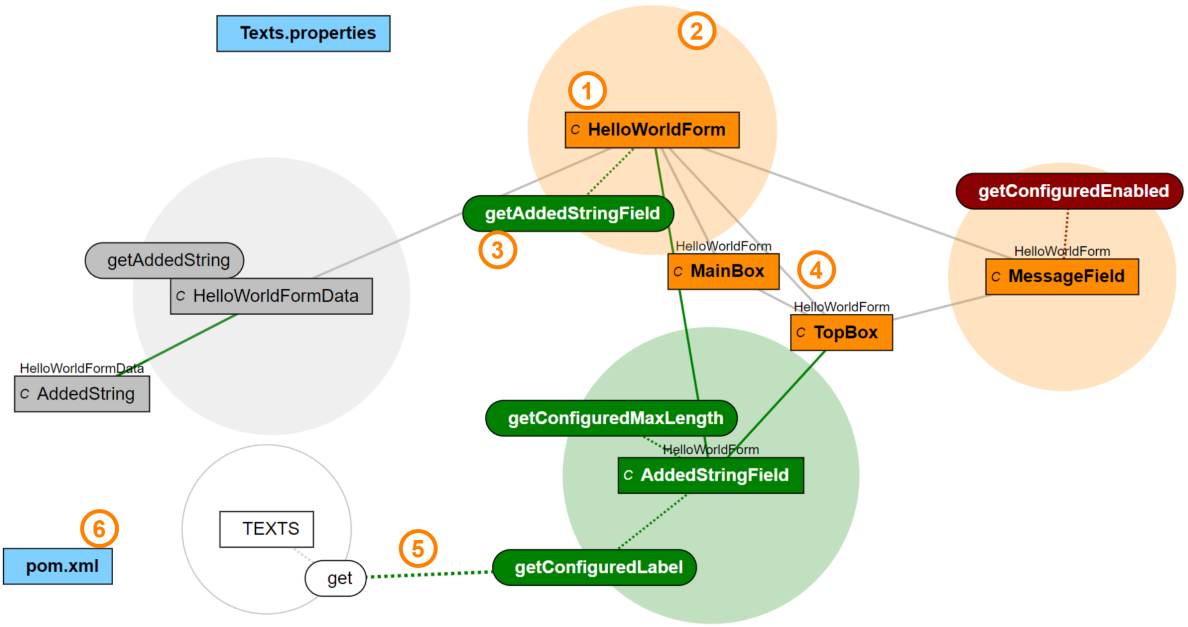
Please write your answer here:

This is a short overview about the developed tool CodeDiffVis (CDV). We will explain to you the visualization and its features step by step.

- CDV works in addition to GitLab merge requests, a web based tool to perform code review on the changes. CDV opens in a separate window and draws the graph according to the changes in the merge request.
- Unfortunately during the survey you will not be able to try the prototype of CDV. Its features will be illustrated through videos. Please keep this into account while answering the questions.
- The code review example is taken from an Eclipse Scout (https://www.eclipse.org/scout/) project. Here is a short preview of how CDV integrates with GitLab:



The *graph* contains nodes that represent classes, interfaces and methods. Links between nodes represent any kind of dependencies (for example, class / interface hierarchies, or method calls).



1. represents a Java class. Classes are denoted with ***C***, Abstract classes with ***A*** and Interfaces with ***I***
2. the circle is added to nodes that contain at least one method in the code change
3. rounded corners of a node indicate a method
4. inner classes are denoted with a label on the top with the corresponding class file name
5. represents a method call. "getConfiguredLabel" of class "AddedStringField" calls "get" from "TEXTS"
6. nodes that do not represent Java components are shown with their full name

- The (change-based) color scheme for nodes and links is: added, changed, deleted, generated, unchanged, non-Java

Please answer the following questions about the information displayed with the *graph* of CodeDiffVis.

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The graph was easy to understand.	◯	◯	◯	◯	◯	◯
The graph displays sufficient information.	◯	◯	◯	◯	◯	◯
In the graph, I thought it was useful to see whether a node is a class, an interface or an abstract class.	◯	◯	◯	◯	◯	◯
In the graph, I would find it useful to see whether a node is public or private.	◯	◯	◯	◯	◯	◯
In the graph, the color-coding of nodes is clear.	◯	◯	◯	◯	◯	◯

Do you have any additional remarks about about the information displayed with the graph? *(Optional)*

Please write your answer here:

Please rate the following statements about the benefits of CodeDiffVis (CDV). *

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
CDV provides information that is not available in any other tool.	◯	◯	◯	◯	◯	◯
With CDV I would be able to understand code changes quicker.	◯	◯	◯	◯	◯	◯
CDV would help me to keep track of my progress during a review.	◯	◯	◯	◯	◯	◯
With CDV I could orient myself better in the merge requests.	◯	◯	◯	◯	◯	◯
With CDV I would find it easier to navigate through the changes.	◯	◯	◯	◯	◯	◯

Considering the statements above, do you have any additional remarks about CodeDiffVis? *(Optional)*

Please write your answer here:

Please answer the below question about CodeDiffVis.

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
I would like to use this tool.	◯	◯	◯	◯	◯	◯

Final Demographic Questions

All questions in this section are optional.

What is your nationality?

Please write your answer here:

What is your gender?

Please choose **only one** of the following:

- ◯ Male
- ◯ Female
- ◯ Prefer to not disclose
- ◯ Prefer to self-define

How old are you?

Please write your answer here:

years

Your working experience as a developer in a professional setting; e.g. as an employee of an IT-company or as an Open-Source contributor.

Please choose the appropriate response for each item:

	None	1 year or less	2 years	3-5 years	6-10 years	11 years or more
For how many years have you developed software in a professional setting?	◯	◯	◯	◯	◯	◯
For how many years have you developed Java software in a professional setting?	◯	◯	◯	◯	◯	◯
For how many years have you performed code review in a professional setting?	◯	◯	◯	◯	◯	◯

How often do you do programming or code reviews?

Please choose the appropriate response for each item:

	Never	About once a year	About once a month	About once a week	Daily or more often
How often do you currently do programming?	◯	◯	◯	◯	◯
How often do you currently do code reviews?	◯	◯	◯	◯	◯

How many hours per week, on average, do you spend on programming?

Please write your answer here:

hours

How many hours per week, on average, do you spend on reviewing code of other developers?

Please write your answer here:

hours

What is you current occupation? (Multiple answers are possible)

Please choose **all** that apply:

- ☐ Professional Developer
- ☐ Academic Researcher
- ☐ Industrial Researcher
- ☐ Project Manager
- ☐ Spare-time Developer
- ☐ Student
- ☐ Other:

General Feedback

Thank you for filling in all answers so far. You are almost done.

All you data is anonymous! We will analyze it, may we also share it in a public research dataset?

(Selecting *yes* allows other researchers and the public to benefit from your answers and effort)

*

Please choose **only one** of the following:

◯ Yes

◯ No

Do you have any final remarks about CodeDiffVis or this survey? *(Optional)*

Please write your answer here:

Would you like to be contacted to try our tool CDV? *

Please choose **only one** of the following:

◯ Yes

◯ No

Would you like to be contacted for an interview as follow-up of this survey? *

Please choose **only one** of the following:

◯ Yes

◯ No

Please leave the email address at which you would like to be contacted.

*

Please write your answer here:

Thank you very much for participating!

If you have any further remarks or questions, please contact the author (mailto:josua.froehlich@uzh.ch) of the survey.

01.08.2020 – 10:00

Submit your survey.
Thank you for completing this survey.

In the video below, you see the layout features of CDV:

- *Zoom* in and out of the graph
- *Drag* single nodes
- Nodes and spheres do not overlap on *collision*
- Connected nodes are highlighted on *hover*

Please evaluate the following claims on CodeDiffVis' layout features:

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The layout features are easy to understand	◯	◯	◯	◯	◯	◯
The layout features seem to be useful for a code review	◯	◯	◯	◯	◯	◯

Do you have any additional remarks on the layout features of CDV? *(Optional)*

Please write your answer here:

In the video below, you see the *graph-to-code* features of CDV:

- Clicking on a node *jumps to the line* of source code
- Not yet clicked nodes are highlighted with a thicker border to *track the progress* of the review

Please evaluate the following claims on CodeDiffVis' graph-to-code features:

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The graph-to-code features are easy to understand	◯	◯	◯	◯	◯	◯
The graph-to-code features seem to be useful for a code review	◯	◯	◯	◯	◯	◯

Do you have any additional remarks on the *graph-to-code* features of CDV? *(Optional)*

Please write your answer here:

In the video below, you see the *code-to-graph* navigation features of CDV:

- Hovering over a line of code in the source code *jumps to the node* and highlights its neighbors

Please evaluate the following claims on CodeDiffVis' code-to-graph features:

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The code-to-graph features are easy to understand	◯	◯	◯	◯	◯	◯
The code-to-graph features seem to be useful for a code review	◯	◯	◯	◯	◯	◯

Do you have any additional remarks on the *code-to-graph* features of CDV? *(Optional)*

Please write your answer here:

In the video below, you see the graph customization features of CDV:

- *Add connected nodes,* i.e. class references or method calls (shift + click on node to expand)
- *Remove nodes* arbitrary, i.e. to track the progress on already reviewed nodes (rightclick on node to remove)

Please evaluate the following claims on CodeDiffVis' graph customization features:

*

Please choose the appropriate response for each item:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	I don't know
The graph customization features are easy to understand	◯	◯	◯	◯	◯	◯
The graph customization features seem to be useful for a code review	◯	◯	◯	◯	◯	◯

Do you have any additional remarks on the customization features of CDV? *(Optional)*

Please write your answer here: