

Visualisation of Code Changes for Code Review

Version of July 1, 2019

Lorenzo Gasparini

Visualisation of Code Changes for Code Review

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Lorenzo Gasparini
born in Spilimbergo, Italy



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Visualisation of Code Changes for Code Review

Author: Lorenzo Gasparini
Student id: 4609905
Email: gasparini.lorenzo@gmail.com

Abstract

Code reviews are a widely adopted practice in software engineering that is proven to increase the quality of the code. Despite its evolution in the last decade, it still presents a number of challenges, such as understanding the changeset in review. In this thesis we research the usage of Software Visualisation paradigms to aid reviewers in the change-understanding process with a tool-based approach.

Based on a survey of the code change visualisation and code navigation research areas, we devise a set of candidate prototypes of a cognitive support review tool, which we iteratively refine involving developers in the process. Through an online survey, we select one of them and build CHANGEVIZ, the implementation of our novel code review environment.

The effectiveness of our approach is validated with a preliminary experiment in which developers perform change-review tasks in our review environment. The results suggest that the features incorporated by our tool are valuable for reviewers.

Thesis Committee:

Chair: Dr. A. Vilanova, Faculty EEMCS, TU Delft
University supervisor: Dr. A. Bacchelli, Department of Informatics,
University of Zurich
External supervisor: M. Eng. T. Baum, Fachgebiet Software Engineering,
Leibniz Universität Hannover
Committee Member: Dr. Mauricio Aniche, Faculty EEMCS, TU Delft

Preface

This thesis concludes my MSc studies in Delft. It has been a learning experience, that made me grow both from a personal and a professional perspective.

I would like to thank my supervisor, Alberto, for overseeing my research and guiding me through the intricacies that are involved with writing a scientific essay. Thanks also to my co-supervisor Tobias, that provided insightful and extensive feedback on my work, and pushed me to achieve my best. And, of course, thanks to Anna and Mauricio for being part of my graduation committee.

Thanks to all my friends, to those who proved that distances do not matter and to all the awesome people that I shared courses and group projects with.

Thanks to my girlfriend, Marta, for bearing with me in the good and the bad times, and reminding me what are the important things in life.

Finally, thanks to my family for their unconditional support and for believing that I could make it. Thanks to my brother who was there to remind me who I am, when I had forgotten it. Thanks to my mother for all the love she has been giving me. And thanks to my father, who passed away during the writing of this thesis, for being proud of me at all times, even when I was not. This thesis is dedicated to him.

Lorenzo Gasparini
Delft, the Netherlands
July 1, 2019

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
2 Background and Related Work	3
2.1 Modern Code Review	3
2.2 Tool-based Cognitive Support Approaches	5
2.3 Code Change Visualisation	7
2.4 Code Layout and Navigation	9
3 Prototyping Phase	13
3.1 Creation	13
3.2 Refinement	19
3.3 Selection	22
3.4 Threats to Validity	24
4 CHANGEVIZ: The Implementation	27
4.1 Interface	27
4.2 Infrastructure	30
4.3 Discussion	31
5 Evaluation	33
5.1 Methodology	33
5.2 Quantitative Results	35
5.3 Qualitative Results and Discussion	36

CONTENTS

6	Conclusions	39
6.1	Contributions and Limitations	39
6.2	Conclusions and Future Work	40
	Bibliography	41
A	Surveys and Results	47
A.1	Low-fidelity Prototype Evaluation: SUS	47
A.2	High-fidelity Implementation Prototype Selection	48
A.3	Tool Evaluation: Control Group	53
A.4	Tool Evaluation: Treatment Group	56
A.5	Tool Evaluation Results	58

List of Figures

2.1	Retrofit pull request #2789 [1] visualised on GitHub	4
2.2	Code change visualisation approaches overview	8
2.3	Code layout and navigation approaches overview	10
3.1	Research method applied for the prototyping phase	13
3.2	Prototype 1, Structural overview	14
3.3	Prototype 1, Callgraph exploration	15
3.4	Prototype 2, Structural overview	16
3.5	Prototype 2, Context exploration	17
3.6	Prototype 3, Base view	18
3.7	Prototype 3, Callgraph exploration	19
3.8	SUS scores, selection phase	23
4.1	CHANGEVIZ interface	28
4.2	Callers sidebar	29
4.3	Callees sidebar	29
4.4	Caller display modal window	30
4.5	Infrastructure overview	31
5.1	Changes to 'FileDialogConfiguration.java' in task 2, in CHANGEVIZ . . .	34
5.2	Task 1 completion times, per group	36

List of Tables

3.1	Average prototypes' SUS scores, refinement phase	22
3.2	p -values of the t -test of the prototypes' SUS scores, in pairs	23
5.1	Task 2 results, per group	36

Chapter 1

Introduction

Code review, particularly in its change-based variant [2], is a widely adopted practice in software engineering, being part of the standard development workflow in open-source projects [3] and in the industry [4]. It has been proven to increase the quality of the code [5] and facilitate knowledge transfer in teams [6].

Despite the evolution of the process in the last decade, code review is still considered to be a challenging task [7]. This is especially true for big changesets [8]. A number of tool-based approaches have been proposed over the years to support the reviewers, including the automatic decomposition of the changeset [9], its textual summarisation [10] and the identification of systematic changes [11].

Among the research directions for tool-based reviewer support, there is the *visualisation of the changes* from a high-level perspective and the incorporation of *IDE-like navigation* features within the review environment [8]. These are the two aspects that we set to explore in our research.

Some code change visualisation approaches have emerged over the last years [12] [13], but there is still relatively little research on the integration of these features into the code review environments [14]. Navigation support has recently been incorporated into some of the popular code review tools [8], but a thorough evaluation of its usefulness is missing.

The goal of this work, therefore, is twofold: providing the implementation of a code review tool that incorporates the code change visualisation and navigation features, and exploring its effectiveness in a real-world usage scenario.

To achieve the first goal, we designed a research process in four steps: 1) creation of low-fidelity interface prototypes, 2) refinement of the prototypes, 3) selection of a prototype for implementation and 4) implementation of the tool.

First, we design three low-fidelity interface prototypes for our tool, incorporating features from previous research. We then use the RITE [15] methodology to refine our prototypes, interviewing developers to assess the perceived effectiveness and usability of the interface. In each interview we guide the user through the design of the interface and obtain comments in two ways: 1) using the think-aloud protocol [16] to obtain rich unstructured feedback and 2) asking the developers to

fill a SUS (System Usability Scale) [17] questionnaire at the end of each interview. The feedback is then used to improve the interface, and the process is repeated until no major usability problems are detected. Afterwards, we design an online survey to select one of the three candidate prototypes for a high-fidelity implementation. Through the survey we collect the usability rating of each prototype, which the users are asked to provide after watching a video walkthrough describing the interface, as well as free-form comments that the users are allowed to submit. With the usability scores obtained and the comments provided by the participants, we then select one of the prototypes and proceed with its implementation.

For the implementation of the high-fidelity prototype, we leverage web technologies and create an online web application, named *CHANGEVIZ*, that works with any pull request on GitHub [18]. In the web-based tool, users can enter the URL to a pull request and perform its inspection leveraging the navigation features offered by our code review environment.

To evaluate *CHANGEVIZ*, we designed an experiment, involving software developers, in which we compared its effectiveness against the standard GitHub review environment, on a set of purpose-built pull requests to a Java software code base. By tracking the task completion time and collecting developers' feedback on the usage of the tool, we explored its usefulness and obtained a set of recommendations for further research in the field.

Structure of the thesis. In Chapter 2 we provide the background to our research, including a description of the modern code review process and a selection of related papers to contextualise our work. Chapter 3 describes the prototyping and refinement phases that led us to the realisation of the tool. In Chapter 4, we present *CHANGEVIZ*, showing some examples of its real-world usage with a number of pull requests from selected projects on GitHub. A preliminary evaluation of its effectiveness is contained in Chapter 5, along with a discussion of the obtained results. Finally, in Chapter 6 we conclude the thesis summarising our contributions, highlighting the limitations of our approach, and proposing directions for future work.

Chapter 2

Background and Related Work

This work is focused on the improvement of software code reviews. In particular, our aim is to build a better review interface, in an effort to provide reviewers with cognitive support to perform this often complex and mentally demanding task. In this chapter we contextualise our work, giving an overview of the modern code review process and presenting the previous research efforts in tool-based support for understanding code changes. The chapter ends with a focus on the previously-researched code change visualisation and navigation approaches, which provided inspiration for our review tool.

2.1 Modern Code Review

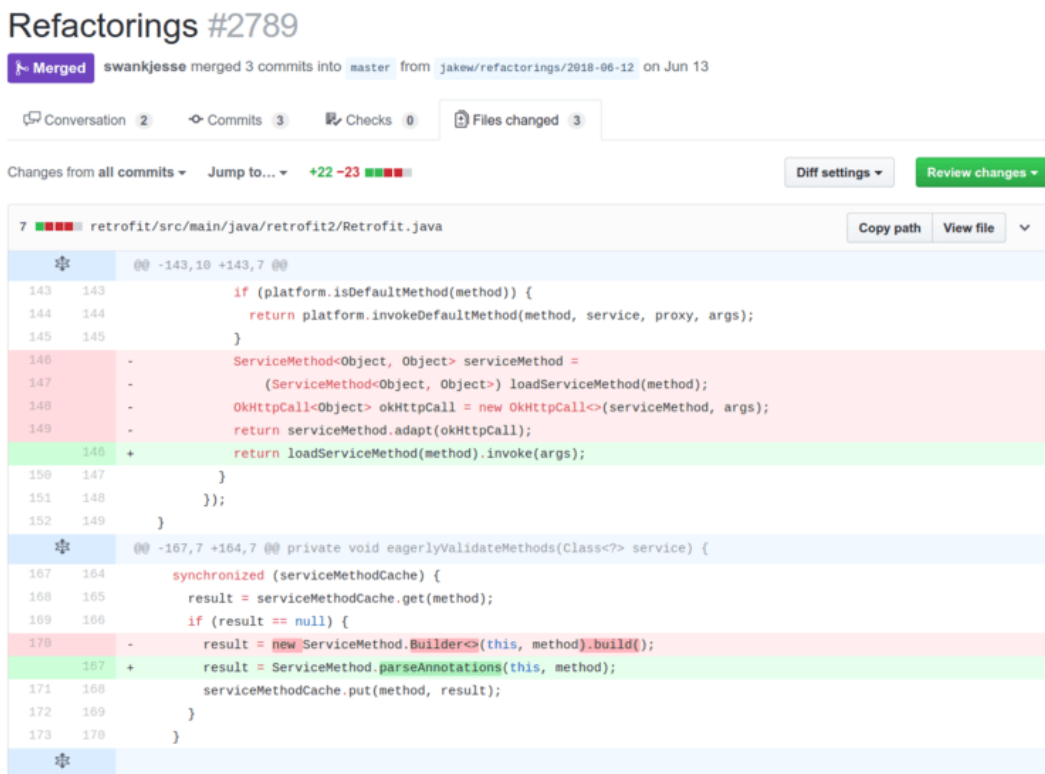
The first formalisation of the code review process is due to Fagan [19], who in 1976 delineated the characteristics of *inspections*. He described it as a thoroughly structured process based on in-person meetings, in which the members of a development team, each with a well-defined role, perform a rigid sequence of distinct operations. The main benefit portrayed in Fagan’s work is the early identification of software defects in the development process, which leads to significant savings in resources.

Over the last two decades, formal code inspections have been replaced by many development teams with a more lightweight, informal and tool-based process: *modern code review (MCR)* [2, 4]. This is the leading style of code review both for companies such as Microsoft [4] and Google [20], and for open source organisations using online collaborative development platforms [3]. In contrast to Fagan’s inspections, this process is often asynchronous, allowing geographically and temporally distributed development teams to collaborate effectively.

The MCR process typically entails the following steps. As soon as a developer has a *changeset* (set of code changes, also known as “patch set”) ready and wants to integrate it into the main software repository, they submit it to the review tool for inspection. The inspection is performed by one or more people different than the changeset’s author, the *reviewers*, who can be manually chosen by the author

2. BACKGROUND AND RELATED WORK

or picked automatically by the tool using a reviewer recommendation algorithm [21, 22]. Often, reviewers are also allowed to pick by themselves a changeset to review. Next, they can inspect the changeset, usually represented by its textual *diff* [23]. They can provide feedback to the author by commenting specific lines of the changeset, consulting him or her about the details of the implementation. The author has the chance to reply and to submit a new version of the changeset, possibly to address reviewers' concerns. Finally, the set of changes is either accepted and integrated into the main repository, or discarded. A number of tools emerged over the last years to support this process [14], both proprietary and open-source, mostly sharing the same characteristics. These include CodeFlow (developed by Microsoft), Gerrit (Google), Differential (Facebook) and Upsource (JetBrains). On-line collaborative development platforms, such as GitHub (Figure 2.1), similarly feature the ability to compare changes and discuss the code.



The image shows a GitHub pull request titled "Refactorings #2789" which has been merged. The diff view shows changes to the file `retrofit/src/main/java/retrofit2/Retrofit.java`. The diff is split into two sections. The first section shows a change to the `invoke` method, where a `return` statement is added and the previous `return` is removed. The second section shows a change to the `eagerlyValidateMethods` method, where a `result` variable is updated to the result of `ServiceMethod.parseAnnotations`.

```
7 retrofit/src/main/java/retrofit2/Retrofit.java
@@ -143,10 +143,7 @@
143 143         if (platform.isDefaultMethod(method)) {
144 144             return platform.invokeDefaultMethod(method, service, proxy, args);
145 145         }
146 -         ServiceMethod<Object, Object> serviceMethod =
147 -             (ServiceMethod<Object, Object>) loadServiceMethod(method);
148 -         OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args);
149 -         return serviceMethod.adapt(okHttpCall);
146 +         return loadServiceMethod(method).invoke(args);
150 147     }
151 148     });
152 149 }
@@ -167,7 +164,7 @@ private void eagerlyValidateMethods(Class<?> service) {
167 164     synchronized (serviceMethodCache) {
168 165         result = serviceMethodCache.get(method);
169 166         if (result == null) {
170 -             result = new ServiceMethod.Builder<>(this, method).build();
167 +             result = ServiceMethod.parseAnnotations(this, method);
171 168             serviceMethodCache.put(method, result);
172 169         }
173 170     }
```

Figure 2.1: Retrofit pull request #2789 [1] visualised on GitHub

While the main motivation behind Fagan's inspections was the early identification of software defects, nowadays reviews are done for a wide number of reasons. In a study [20] on the code review practices at Google, Sadowski et al. identified multiple motivations for performing reviews: while finding defects is a welcome addition, they are first of all a way to ensure the maintainability and understandability of the code. The preservation of design and style consistency, and the ability

to retrospectively analyse the evolution of a code change, are some of the other reasons that they identified. Similarly, studying code reviews at Microsoft, Bacchelli and Bird [4] found that reviews were less than expected about finding bugs, highlighting the importance of other aspects such as the creation of alternative solutions, the increase of team awareness and the transfer of knowledge.

The biggest challenge that reviewers face in their task is understanding the changeset under review [7]. This is especially true for large changesets [8]. As noted by Bacchelli and Bird [4], despite the vast amount of research in code comprehension, the most popular code review tools portray the changeset with a simple syntax-highlighted diff, offering little to no additional support to comprehend the artifact under review. Baum and Schneider [8] proposed a number of tool features to support the understanding process. Identifying the determination of a change's risk as an important concern to understand it Tao et al. [24] similarly provided a set of desired tool characteristics to help determine its impact on other components of the code base.

2.2 Tool-based Cognitive Support Approaches

In this section we present an overview on the proposed code review tool features to support the understanding of a changeset.

Change Summarisation. A number of methods have been researched to create a textual summary of the changes between two versions of a program. Buse and Weimer [25] proposed *DeltaDoc*, a technique for the automatic generation of human-readable descriptions of program differences. Their approach, based on symbolic execution and code summarisation, documents the impact of a changeset on the runtime behaviour of the program. Similarly, Cortes-Coy et al. designed *ChangeScribe* [10], a technique to automatically generate commit messages of Java repositories based on fine-grained differences, showing that their generated messages are preferred in most cases to the ones originally written by their author. A mixed approach named *ARENA* [26] was presented by Moreno et al., combining structural code changes summaries, documentation, libraries and licenses changes and linking them to information mined from the issue tracker to generate release notes. Observing that human-written commit messages typically have only one sentence, Jiang and McMillan [27] proposed an alternative technique to generate one-phrase summaries based on natural language processing.

Change Untangling. The decomposition of code changesets that address multiple development concerns, called *composite changesets*, has been shown to support reviewers in their tasks [28]. Herzig and Zeller [29], analysing a number of open-source Java projects, found that up to 16.5% of the bug fixes consisted of composite changes. They proposed a prototype of untangling algorithm based on the separation of a changeset into a set of change operations, aimed at reducing the noise encountered when mining software repositories. Barnett et al. [9] presented and evaluated the effectiveness of *CLUSTERCHANGES*, a technique for automatically

2. BACKGROUND AND RELATED WORK

decomposing changesets in the context of code reviews. The tool builds a partitioning of the diff-regions in the changeset, based on def-use relationships, and displays it in a tree-like view. An approach that incorporates multiple heuristics to perform the partitioning, including the pattern of the changes and their static dependencies, is the one proposed by Tao and Kim [30]. Evaluating the code review performance of their study’s participants by tracking the task completion time on tangled and untangled changesets, the authors found an increase in the review effectiveness when changes are partitioned according to their algorithm. Dias et al. [31] contributed with a public dataset of untangled changes and by expanding the work done by Herzig and Zeller with *EpiceaUntangler*, a novel approach integrated with the IDE that tracks fine-grained code changes and proposes, based on the tracked events, an automatically computed partitioning whenever the developer is about to commit a changeset. An interactive tool named CHGCUTTER was proposed by Guo and Song [32]. Their technique computes subsets of related code changes basing on static analysis and generates intermediate versions of the source program, each corresponding to a subset of changes, that are guaranteed to compile and on which test cases can be run.

Identification of Systematic Changes. Systematic changes between two versions of a program’s source code are typically not easy to identify using the standard diff-view, complicating the change-understanding process. This is the case for code refactorings, which “pollute” the diff with changes that do not affect the program behaviour. To solve this problem, Kim and Notkin proposed *LSdiff* [33, 34], a tool that captures structural differences and infers logical rules which are then used to represent the differences, possibly along with anomalies indicating deviations from the rule. By assessing the benefits of their approach with a focus group, the authors find that their representation of systematic changes can be a valuable complement to the diff. Building on the prior work by Kim and Notkin, Prete et al. proposed REF-FINDER [35], an approach that is able to identify complex refactoring instances between two versions of a program, basing on a catalogue of template refactoring rules. In a case study on an open-source project, REF-FINDER achieved a precision of 0.79 in the identification of refactorings. Kawrykow and Robillard proposed DIFFCAT [36], a technique to identify “non-essential” modifications in software changes, such as cosmetic and behaviour-preserving differences. Similarly, Thangthumachit et al. [37] argued the importance of separating the refactorings from the source code differences, and presented an approach to produce program differences excluding the refactorings effects. In contrast to DIFFCAT, which can only detect renames, their technique supports a wide range of refactorings. The authors later extended their work with *REdiffs* [38], an interface for analysing changes with the ability to separate refactoring effects. CRITICS [11] is an interactive approach that, given a specific code region within a changeset, builds a change template which is matched against the code base. The matching results in a summary of the systematic edits, along with potentially inconsistent edits, and can be interactively tuned by the developer by customising the change template. By performing a user study with professional developers, the authors of the tool found

that most of them would like to have its features integrated into their code review environment.

Change Impact Analysis. Determining the risk of a change, and identifying the portions of code that are impacted by it, is one of the major concerns in the change-understanding process, being at the same time information difficult to acquire [24]. Reviewers typically have to manually inspect all the dependencies of the changed parts of code to determine a change's risk. *JRipples* [39] is a tool that assists software change by identifying the classes impacted by a change, given an initial set of classes to begin the impact propagation. Ryder et al. [40] proposed *Chianti*, which similarly to *JRipples* is implemented within the Eclipse IDE and works with Java programs. *Chianti* partitions the changeset into a set of atomic changes and computes a many-to-many association between the atomic changes and the test cases that they could possibly affect, allowing the developer to isolate the changes responsible for a test's failure. A different approach, which uses statistical learning instead of static analysis to compute the relationships between the software components of a program source code, is the one proposed by Ceccarelli et al. [41]. Their algorithm uses the Granger causality test to identify co-changes in a software repository's history and, based on the idea that changes to artifacts which co-changed in the past may impact each other, computes an impact graph. As remarked by the authors, impact relationships learned using their method are complementary to the ones learned with traditional static analysis approaches, making the former an interesting approach to determine a change's risk. *ImpactViz* [42] is a visual approach to impact analysis: classes are represented as nodes in a force-directed graph layout, with colours used to encode whether they have been modified, and directed edges representing method calls between them. The effectiveness of the visual approach is validated by Delfim et al. [43], who carried out a controlled experiment comparing developers' effectiveness in estimating the impact of a change using its visual representation and manually inspecting the raw code. Their results support the hypothesis that a change's impact can be more easily identified with visual support.

2.3 Code Change Visualisation

Within the Software Visualisation research field, whose aim is to employ visual paradigms to facilitate software comprehension, a number of Software Evolution Visualisation (*SEV*) approaches have been proposed over the last years. These techniques are typically designed to assist the study and analysis of the evolution of software using visual metaphors. In their mapping study, Novais et al. [44] distinguish between two categories of SEV approaches: *differential*, focused on the changes between two versions of a program, and *temporal*, supporting the visualisation of a considerable number of versions. As the goal of our work is to create a better interface for code reviews, which typically involve the analysis of the differences between two versions of a software, we focus on the differential approaches.

2. BACKGROUND AND RELATED WORK

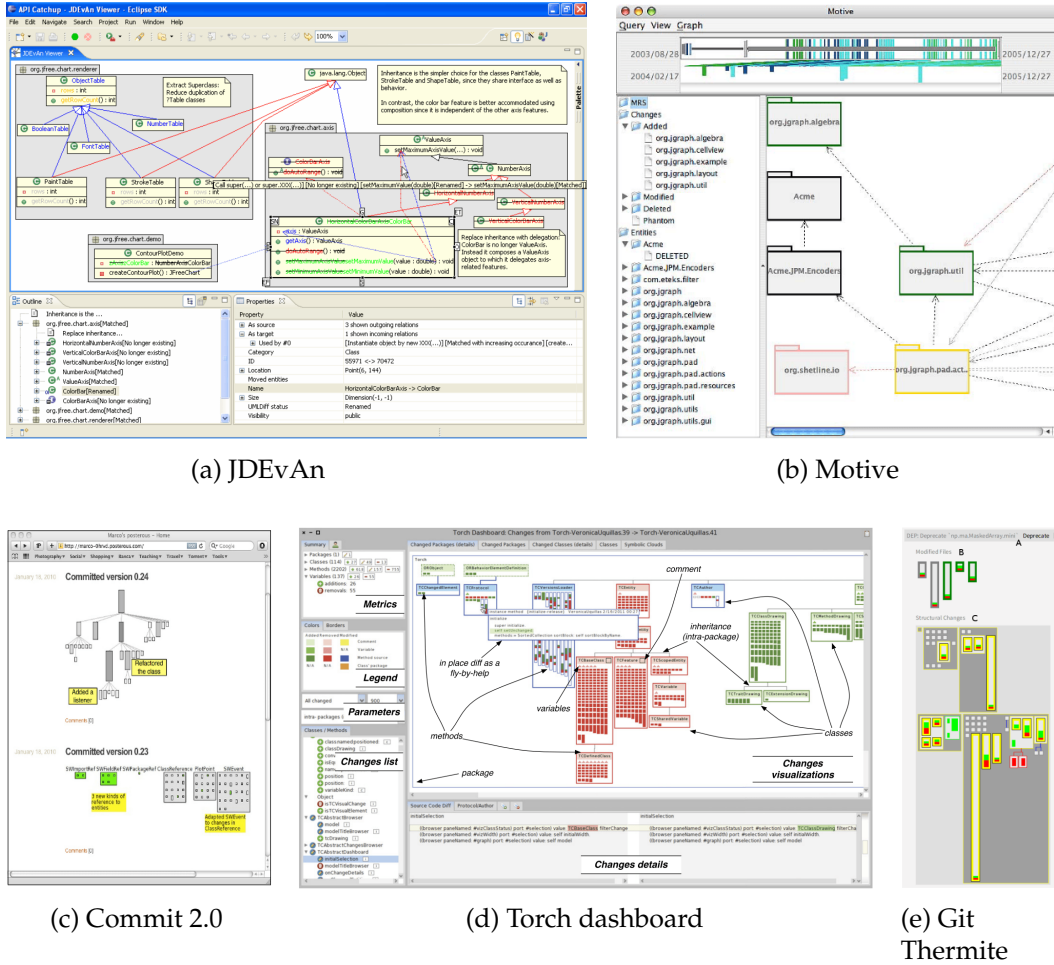


Figure 2.2: Code change visualisation approaches overview

JDevAn [45] is a tool, implemented as an Eclipse IDE plugin, that helps the analysis of the design evolution of Java software. It is based on the UMLDiff algorithm [46], which detects structural changes between two versions of a program, including the additions, removals, renames of packages, classes and methods, and the dependencies between them. Its interface (Figure 2.2a) is divided into three parts: an UML diagram showing the software elements of interest along with the relations between them and their changes, a tree view of the diagram’s elements to ease navigation, and a tab showing the properties of the selected component. In the UML diagram, colours are used to represent the addition, removal and modification of the components and the relations.

McNair et al. proposed *Motive* [47], which similarly to *JDevAn* is targeted at the analysis of architectural changes of Java programs. The differences are portrayed in an “architectural impact view” (Figure 2.2b), a UML-style graph showing the packages or the classes of the system, their dependencies, and the changes encoded

with colours. By using a scanning approach to extract the structural differences, Motive is able to overcome possible syntax errors in the code when performing the extraction, at the cost of a lower precision compared to the usage of parsing.

Commit 2.0 [13] is an IDE enhancement designed to create rich documentation of commits to Smalltalk programs. When the developer is about to commit a change to the code, the tool creates a coarse-grained visualisation of the structural changes as a graph of nested boxes, using colours to encode the type of change. The visualisation (Figure 2.2c) can be enriched with annotations and the addition of fine-grained code differences, before being stored in a blog for persistence.

The “nested boxes” visual metaphor is borrowed and extended by *Torch* [12], an integrated dashboard to assist developers and integrators in understanding changes to object-oriented programs. The tool’s interface (Figure 2.2d) displays the characteristics of a changeset in a number of different ways: presenting a list of metrics about the number of entities (packages, classes, methods) modified, a list of the structural changes, a visualisation of the architectural differences and possibly the details (with a diff-view) of an entity when the user selects one. The main visualisation allows to choose whether to show only the modified entities or all of them, and between different levels of granularity (packages or classes). The “fly-by-help” feature allows the user to inspect the line-level changes of an entity by hovering the cursor on its visual representation.

Git Thermite [48] is a tool inspired by Torch, designed to help integrators using Git and GitHub to understand the internal structure of a pull request. A “business card” (Figure 2.2e) visually displays the characteristics of the changeset, including the metrics and the structural changes, encoded with colours. Like Torch, Git Thermite features a contextual diff that lets the user access the line-level changes of an entity by hovering it with the cursor. Moreover, it supports multiple programming languages, like Python and Smalltalk.

2.4 Code Layout and Navigation

As noted by Tao et al. [24] in their study on change-understanding tasks, developers call for features that allow the navigation of the context of the changeset, such as the “go to definition” and “find references” features commonly found in IDEs. A number of approaches have been proposed over the last years to assist the navigation of the code and its layout on the editing interface, based on the method call relationships.

Relo [49] is a tool for the incremental exploration of Java software code bases, which assists the construction of a mental model for the exploration of the code, reducing the cognitive load for the developer. Starting the analysis from a single code entity, like a package or a class, the user can expand its context following the method call and class inheritance relationships. The entities are represented using UML-style diagrams (Figure 2.3a) and are positioned automatically based on the type of relationship. The zoom feature allows to display and edit a method’s

2. BACKGROUND AND RELATED WORK

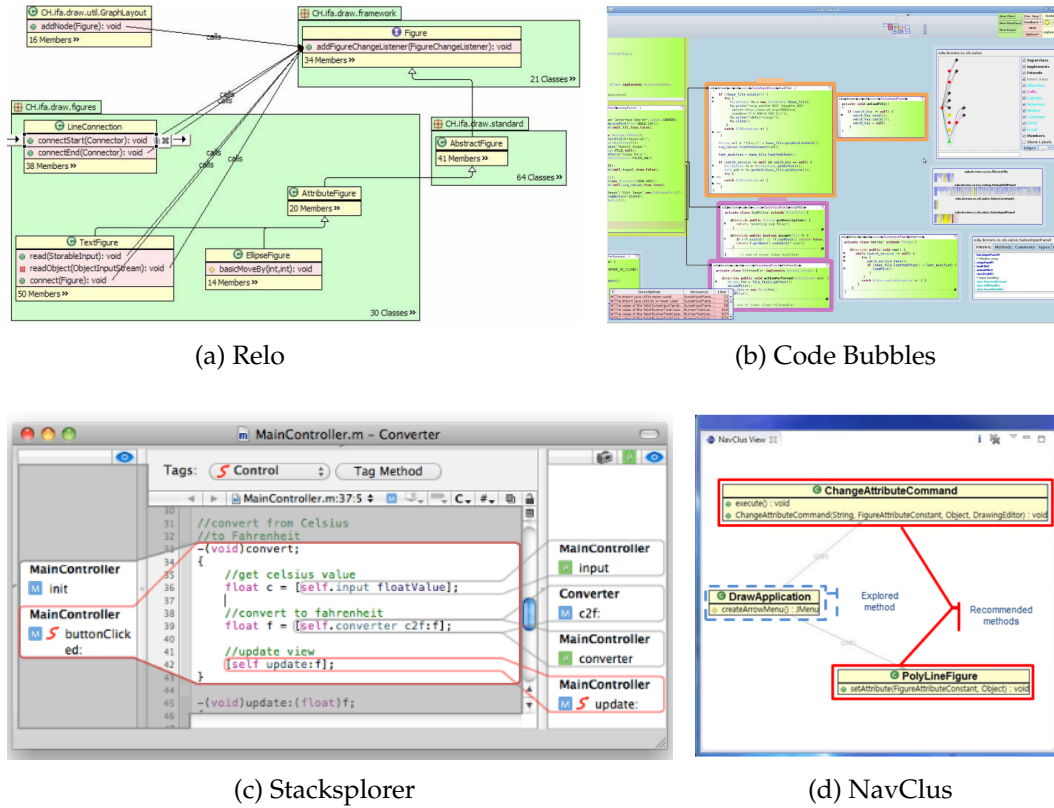


Figure 2.3: Code layout and navigation approaches overview

definition directly within the tool.

A novel interface (Figure 2.3b) for understanding and editing code is the one proposed by Bragdon et al. [50] with *Code Bubbles*. The tool is based on the “bubbles” visual metaphor: the code is arranged in editable fragments which do not overlap, but push each other away, and can be manually positioned by the user. The bubbles are connected by edges that represent relationships among them, such as method calls, and influence the way in which they are laid out on the canvas. Instead of working at the files granularity, as common IDEs do, Code Bubbles works at the method granularity, which the authors argue is a more spatially efficient way to view and edit code. Similarly to Relo, it allows the incremental exploration of a method’s context, with the “Open Declaration” feature.

Stacksporer [51] is a tool designed to assist the navigation of the code in IDEs based on the callgraph, implemented as a plugin to Xcode. It enhances the IDE’s interface with two sidebars (Figure 2.3c) that display at all times the callers and the callees of the currently focused method. The hypothesis that having the callgraph always available helps retain the context of a method and assists its navigation is supported by a quantitative study performed by the authors, in which developers using the tool perform better than the ones not using it on a set of code maintenance

tasks.

Lee et al. [52] proposed *NavClus*, an approach for the graphical recommendation of interesting portions of code. The tool mines the daily interactions between the developer and a code base in the IDE, and recommends in a separate window (Figure 2.3d) relevant methods based on the current context. The user can click on one of the recommended methods to bring up the corresponding definition in the IDE.

In this chapter we described the modern code review process and presented a number of tool-based research approaches that have been developed in the last years to tackle the challenges faced by reviewers. Next, we will focus on the research process that led to the creation of a low-fidelity prototype of our own code review environment, incorporating the *code change visualisation* and *code layout and navigation* aspects based on the previous research efforts in these areas.

Chapter 3

Prototyping Phase

In this chapter we describe the process that led to the creation of a low-fidelity prototype of CHANGEVIZ. The research method employed in this phase is outlined in Figure 3.1.

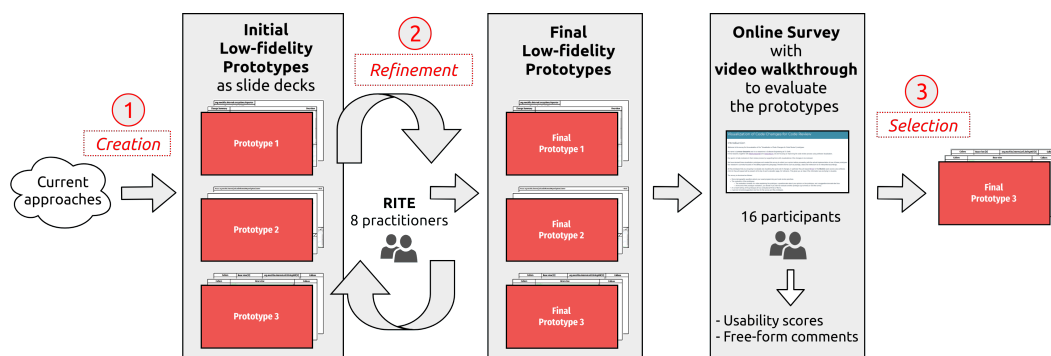


Figure 3.1: Research method applied for the prototyping phase

First, we present the creation phase (Point 1) during which we built a set of three candidate prototypes based on the state of the art in code change visualisation and code navigation research. We proceed illustrating the refinement step (Point 2) in which we used the RITE [15] technique to refine our set of prototypes, conducting face-to-face interviews with developers. Finally, we describe the selection phase (Point 3) which involved an online survey to evaluate the refined interfaces and led to the choice of one of them for a high-fidelity implementation.

3.1 Creation

The first step in the research process was the creation of a set of low-fidelity prototypes of our tool's interface. By creating first a low-fidelity version of them, in contrast with a high-fidelity implementation, we were able to iterate and integrate the external feedback to the interface more rapidly.

3. PROTOTYPING PHASE

We focused our research on objected-oriented languages, and more specifically Java. Terms such as ‘package’, ‘class’ and ‘method’ should be interpreted accordingly. As a reference changeset to visualise, we picked pull request #1247 [53] to Mockito, a mocking framework for Java, which included multiple types of structural changes (modification and deletion of classes and methods) that showcased our prototypes’ features.

Based on the research approaches that we reviewed, we devised 3 prototypes incorporating different navigation and visualisation features. This allowed us to evaluate the effectiveness of these features, in the context of code reviews, independently from each other. The number of prototypes was the result of a trade off between creating a single prototype, that could not have incorporated all the features that resulted from our research survey without overcomplicating the interface, and a large number of prototypes that would have introduced significant challenges in the selection process. Each prototype consisted in a storyboard, in the form of a Google Slides [54] presentation, portraying the interface of the tool and a sequence of realistic interactions with it.

3.1.1 Prototype 1

The first prototype review environment is comprised of two main views: the *structural overview* and the *callgraph exploration* view. Throughout the interface, a uniform set of colours is used to encode the modification status of an entity (package, class or method): green for the newly added entities, blue and red for the modified and removed ones respectively, and black for those that were not modified.

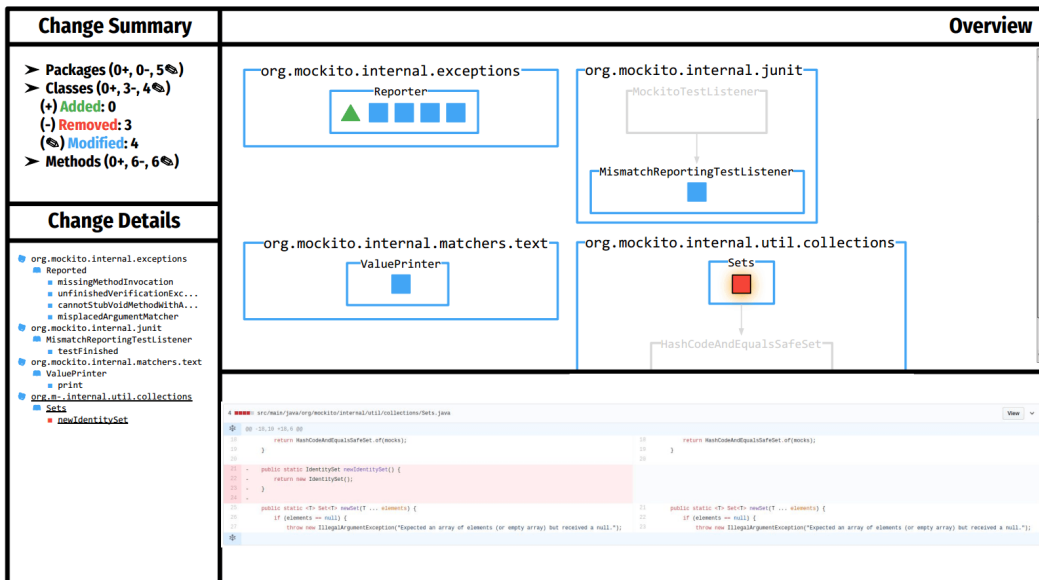


Figure 3.2: Prototype 1, Structural overview

The structural overview (Figure 3.2) is inspired by the Torch dashboard [12] and similarly provides a recap of the structural changes introduced by the changeset under analysis, with three interconnected perspectives. The ‘change summary’ perspective on the top left provides metrics regarding the number of added, removed and modified packages, classes and methods. It also acts as a global filter for the overview, allowing the user to focus only on a subset of the entities affected by the changeset when the corresponding category is selected; in the example in Figure 3.2, for instance, only the modified classes are shown. On the bottom left, the ‘change details’ perspective lists the changed code entities, grouped by class and by package. The ‘visual overview’ on the top right displays a visual summary of the changeset in which the changed packages, each represented by a coloured outline along with the name of the package, are laid out automatically on a vertically-scrollable canvas. Within each package, the classes that were modified therein are shown with an outline together with their name. In turn, the classes contain coloured squares that represent their modified methods, if any. Whenever a class is part of a class hierarchy, its immediate superclasses and subclasses are displayed in grey, contextualising it within the hierarchy. Each entity can be selected by clicking on it either in the visual overview or the ‘change details’ perspective, which leads to the display of the underlying source code differences in the bottom right ‘diff’ panel of the interface.

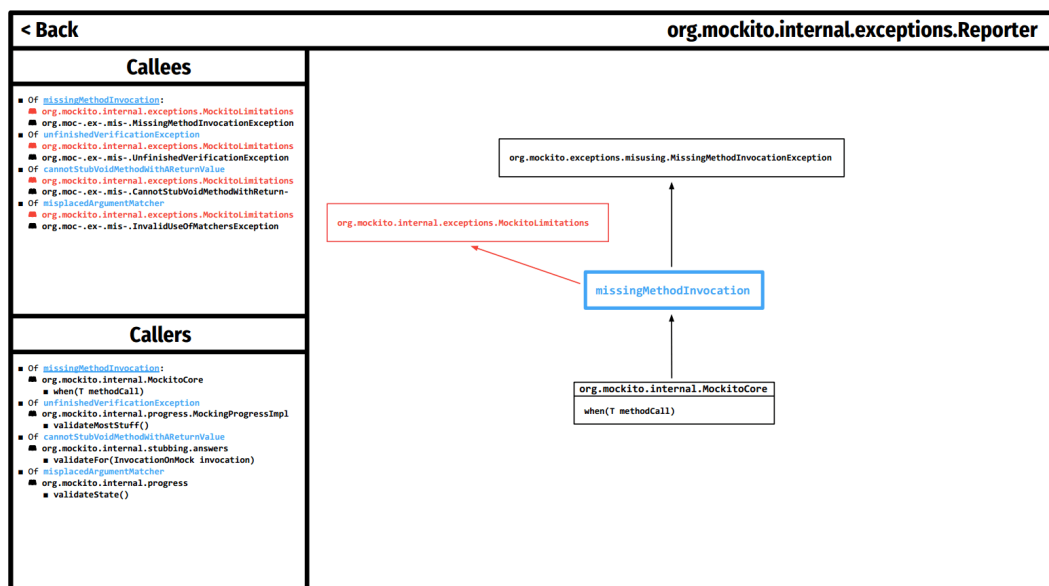


Figure 3.3: Prototype 1, Callgraph exploration

When a modified class or a method is double-clicked, the callgraph exploration view (Figure 3.3) is brought up. This view, inspired by NavClus [52], focuses on a single class and allows the user to explore the neighbours of its modified methods in the callgraph. On the left, a sidebar separately lists the *callees* (methods that

3. PROTOTYPING PHASE

are called by the methods of the class in focus) and the *callers* (methods that call the methods of the class). The user can customise the view by selecting one of the class' methods in the sidebar, which will be therefore shown in the callgraph view on the right. The selected method is placed in the center and the relationships with its callees and callers, both modified and unmodified, are shown by directed arrows pointing towards the destination of the method call. By pressing the 'back' button, the callgraph exploration view is closed and the user is brought back to the structural overview.

3.1.2 Prototype 2

Similarly to the first one, the second prototype is composed of two views: a *structural overview* and a *context exploration* view. Colours are used in the same way all over the interface to express the modification status of the entities.

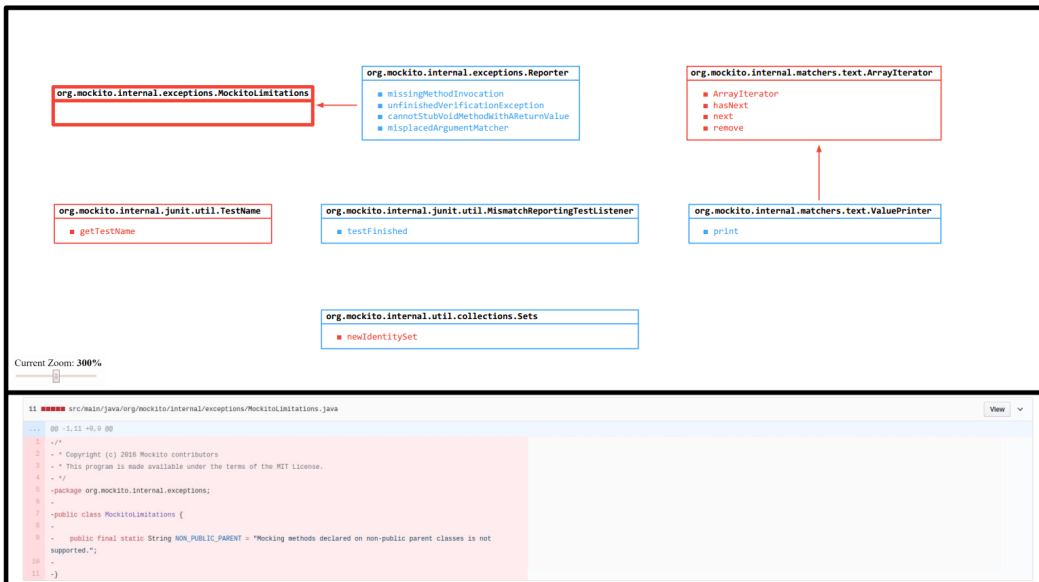


Figure 3.4: Prototype 2, Structural overview

The structural overview (Figure 3.4) takes inspiration from the bubble metaphor used by Code Bubbles [55]: the changes, grouped by Java class, are automatically laid out on a canvas which extends infinitely in every direction. Using the zoom feature, the user can decide to focus on a specific area of the canvas. The classes, represented as UML-style diagrams and listing the modified methods, are connected to each other by directed arrows. A link represents a callee or caller relationship between the methods of two different classes; the layout algorithm takes these relationships into account, placing linked classes next to each other. The underlying source code differences can be examined in two ways: by hovering a class, which brings up an overlay showing the corresponding diff in a compact version

(*unified diff*), or by selecting it, which populates the bottom panel of the view with the full version of the diff (*split diff*). When one of the arrows is hovered, the details of the method calls are displayed in an overlay.

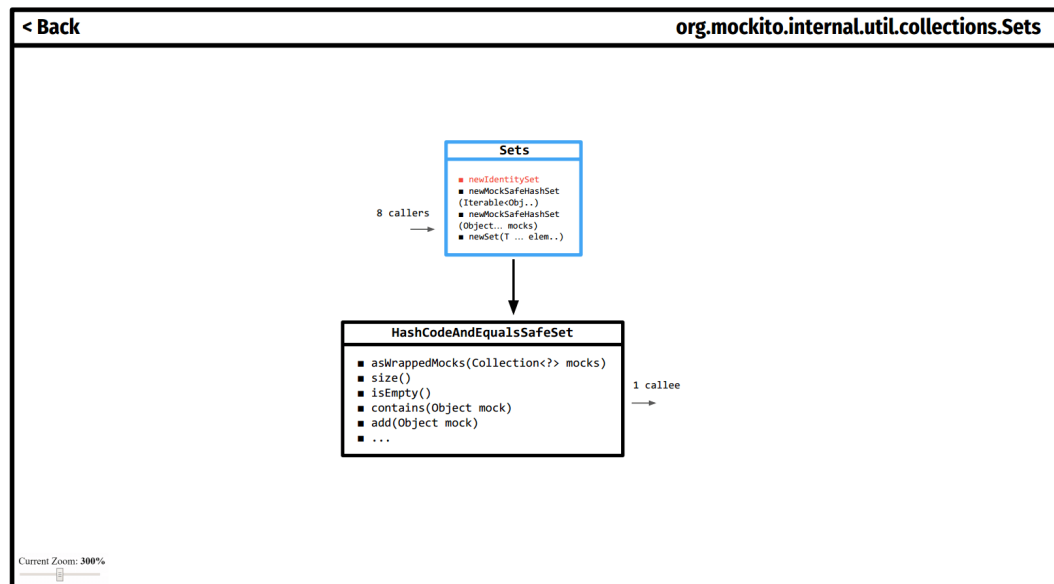


Figure 3.5: Prototype 2, Context exploration

By double-clicking on a class, the *context exploration* feature (Figure 3.5) is activated, and the user is presented with a new view centered on the chosen class. This feature, inspired by Relo [49], allows the incremental exploration of the context of the class following the method call and class hierarchy relationships. The class is represented by an UML-style diagram listing its methods (both modified and unmodified) and is placed at the center of an infinite zoomable canvas. Grey arrows surround the class in focus and indicate the possibilities for context exploration: callers of the methods in the class are placed on the left, callees on the right, superclasses on the top and subclasses on the bottom. Only one arrow is shown for each type of existing related entity, along with a label indicating the number of entities of that kind, in order not to overcrowd the visualisation. When the user clicks on the arrow, or the label, they are substituted by a tree enumerating all the entities with their qualifier. By clicking on the qualifier, the corresponding entity is brought up in the canvas and placed according to its relationship with the main class, linked to it by an arrow. The same context exploration procedure is available for the new entity, allowing the user to construct an exploration path while keeping track of its history. By overlaying an entity, its source code (or diff if the entity was modified) is revealed in an overlay letting the user examine it.

3. PROTOTYPING PHASE

3.1.3 Prototype 3

The third prototype is inspired by Stacksplorer [51] and focuses on enabling an efficient navigation of the context of the changeset. The callgraph navigation features offered by Stacksplorer have been compared to the ones commonly offered by IDEs, and have been found to increase developers' effectiveness in solving programming tasks that required browsing the context of the code [56].

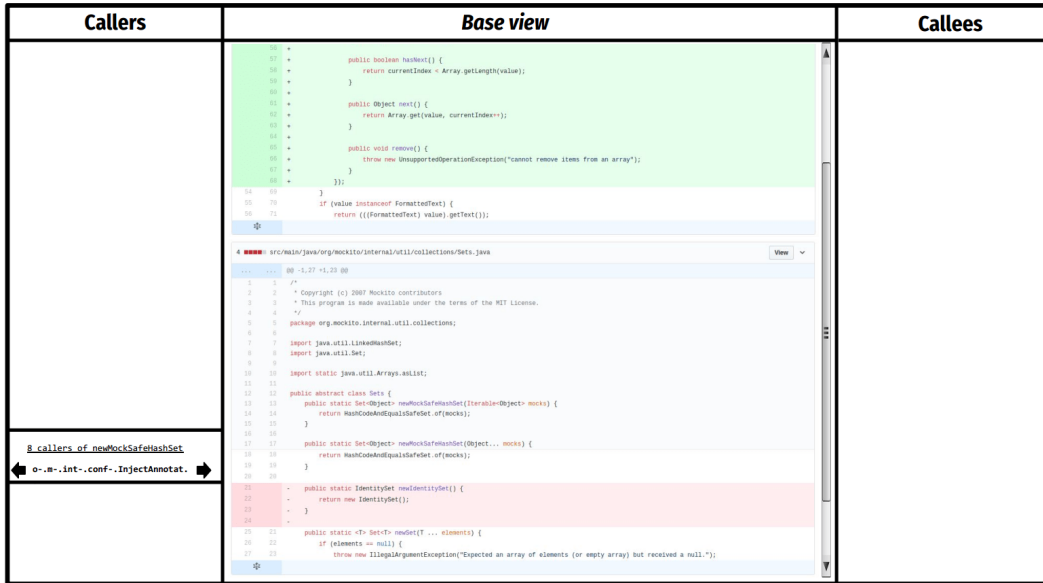


Figure 3.6: Prototype 3, Base view

The base view (Figure 3.6) features the changeset at the center, as unified diff, surrounded by two sidebars. The sidebars allow the user to navigate the context of the diff based on the callgraph: the left one lists the callers of the methods whose declaration intersects the portion of code shown in the diff, while the right sidebar lists their callees. The callers and callees in the sidebars are represented by a box whose top and bottom borders match the start and end line of the corresponding method declaration in the diff: as the user scrolls the code, the sidebars update accordingly, keeping the caller and callee information synchronised to the portion of the changeset currently visualised. Each box in the sidebars lists the number of callers (or callees) of the related method declaration, along with a list of their qualified names; only one caller (callee) is shown at time to optimise the space, while the others can be accessed using two arrows that allow to scroll the list.

When the user clicks on a caller, or callee, the class containing it is opened as a new tab (Figure 3.7). The user can now browse the source code of the class, which is shown at the center of the interface. The sidebars show the callers and callees of the methods of the opened class, enabling the navigation of its context in the same manner as explained above. The horizontal scrollbar allows the user to move between tabs and visualise more than one of them at time, as seen in Visuocode

Callers	Base view [X]	<i>org.mockito.internal.util.StringUtil</i> [X]	Callees
	<pre data-bbox="536 371 861 889"> 1 /* 2 * Copyright (c) 2017 Mockito contributors 3 * This program is made available under the terms of the MIT license. 4 */ 5 package org.mockito.internal.util; 6 7 import static java.util.Arrays.asList; 8 9 import java.util.Collection; 10 import java.util.regex.Matcher; 11 import java.util.regex.Pattern; 12 13 public class StringUtil { 14 15 private static final Pattern CAPS = Pattern.compile("[A-Z\\d]{4,2\\d}"); 16 17 private StringUtil() {} 18 /** 19 * @param text 20 * to have the first line removed 21 * @return text first line 22 */ 23 public static String removeFirstLine(String text) { 24 return text.replaceFirst("\\n", ""); 25 } 26 27 /** 28 * Joins Strings with line break character. It adds line break in front, too. 29 * This makes it something like "format" so really 'join'. 30 */ 31 public static String join(Object... linesToBreak) { 32 return join("\n", asList(linesToBreak)); 33 } 34 35 /** 36 * Joins Strings with EOL character 37 * @param start the starting string 38 * @param lines collection to join 39 */ 40 public static String join(String start, Collection<?> lines) { 41 return join(start, "", lines); 42 } 43 44 /** 45 * Joins Strings with EOL character 46 * @param start the starting string 47 * @param linesCollection the prefix for each line to be joined 48 * @param lines collection to join </pre>		

Figure 3.7: Prototype 3, Callgraph exploration

[57]. The sidebars show at all times the callgraph information of the class that is in focus, which has its tab name italicised. The focus can be switched to another class (or the base view) either by clicking on the name of its tab or by scrolling horizontally until the desired tab occupies more than half of the total width.

3.2 Refinement

Following the creation of the initial set of prototypes, we proceeded with the second step of our research process: their refinement. For this purpose, we employed the RITE technique [15], devised to find problems in the interface and rapidly fix them.

The evaluation of the prototypes was performed by interviewing developers, more specifically postgraduate students, who all had previous experience with Java and the GitHub review interface. We walked them through each prototype's storyboard and obtained comments in two ways: 1) using the think-aloud protocol [16] to acquire rich unstructured feedback and 2) asking the candidates to fill a SUS (System Usability Scale) [17] questionnaire at the end of the walkthrough. The order of the prototypes, as presented to each participant, was randomised in order to avoid biases towards one of them. After each interview, we analysed the free-form comments and the SUS scale results to identify design flaws and areas of improvement, if any, and then modified the prototypes accordingly.

We performed a total of 4 iterations of the RITE technique, interviewing 2 developers on the same set of prototypes at each iteration, after which we reached a stable version that did not contain any major design flaws. Due to a procedural mistake, the RITE process of prototypes 2 and 3 was terminated early, without val-

idating their last iteration. In the following, we will report the feedback received by the interviewees (D1-D8) for each of the prototypes, and the changes we made to the interfaces to address it.

3.2.1 Prototype 1

The first prototype was generally received well by the candidates (e.g., “I like it, it’s clear, it’s user friendly. It’s very visual and you get an immediate idea of the changes in the structure” [D4]) despite some of them expressing concerns regarding its complexity: “It’s distant from the Github pull request view [...] it’s somehow too complex” [D3].

During the first iteration, D2 suggested to enable the selection of multiple modified entities at the same time in the structural overview, allowing to browse their diffs one after the other (“Can you click on multiple entities and see their diffs one after the other? That would be useful”). D1 proposed two modifications: showing an overlay containing the diff when hovering an entity and removing the caller/callee separation in the callgraph exploration view’s sidebar. All these features were integrated in the interface.

On the following iteration, based on the candidates’ feedback we further modified the sidebar in the callgraph exploration view: in addition to the modified methods, the user can also decide to focus on a method of the selected class that was not modified. Another change was made to enable the expansion and contraction of the callers/callees list of each focus method, in order to use space more efficiently.

The last amendment to the interface was made in the third iteration, based on a remark on the structural overview: “I would like to see the class variables represented as well” [D7]. The class variables were consequently added to all three perspectives of the structural overview, representing them with a triangle.

3.2.2 Prototype 2

The second prototype received mixed feedback: while some candidates appreciated its features (e.g., “The display of callgraph and class hierarchy relationships is useful, as is the incremental expansion of the context” [D5]), others feared that it might not be easily adoptable: “It may take some time for the developers to get accustomed to it” [D3].

In response to the candidates feedback, who noted that the context switch between the structural overview and the context exploration view could make them lose the mental model constructed so far, we decided to unify the two views during the first iteration. To accommodate the newly opened entities, the other fragments are rearranged according to an algorithm that minimises their total joint movement, taking the relationships among them into account. Similarly to the first prototype, the selection of multiple entities was added on the suggestion of D2.

On the second iteration, a candidate remarked that the context expansion feature did not support multiple callers/callees: “What happens when you have a lot of callers?” [D4]. The feature was modified so that when the user clicks on the arrow suggesting a possible context expansion, a tree pops up showing all the available possibilities. Basing on the feedback of D3, two other features were added during this iteration. First, a miniature map was added in the bottom left of the interface to help users orient themselves in the canvas, especially in the case of big changesets. Second, a button was added to the bottom right diff panel to enable its maximisation to the entire screen when needed.

During the third iteration, following the candidates’ remarks we added the possibility to close the fragments opened with the context expansion feature (“Once you have expanded the diagram following the suggested directions can you minimize it again?” [D5]).

The last modification was made during the fourth iteration, to address a side effect of the unification between the structural overview and the context exploration view. The candidates noted that expanding the entities’ context could lead to an excessive amount of information displayed on the canvas, making it hard for them to maintain the focus. To solve this, we added the option to double-click on an entity to focus on it: when this happens, all the fragments in the canvas are hidden, except for the one the user clicked and the entities connected to it.

3.2.3 Prototype 3

Most of the candidates appreciated the third prototype due to its similarity to the review interfaces that they were already used to (e.g., “It’s nice because developers are familiar with this view” [D2]). A downside reported by multiple participants was that, in comparison with the other prototypes, this one reported less information: “[...] you can see less information than the other two prototypes, like the structure and class hierarchy” [D4].

On the second iteration, following a suggestion from D3, we modified the call-graph exploration feature so that when the class containing the caller/callee is opened, the code is automatically scrolled to the position of the caller/callee and an arrow indicates the exact line where it is located.

The horizontal scrollbar feature that allows to visualise two tabs next to each other received criticism from multiple participants during the interviews (e.g., “The horizontal bar scrolling is more fancy than useful, you could just select the tab with a click” [D4]) and was therefore removed during the fourth and last iteration.

3.2.4 SUS scores

In Table 3.1 we report the average SUS scores obtained by the prototypes across all the iterations.

The best average score was achieved by prototype 3 with 83.3, empirically associated with the adjective ‘excellent’ [58]. Prototype 1 follows with an average

	Average SUS score
Prototype 1	74.6 (s = 7.1)
Prototype 2	72.2 (s = 7.9)
Prototype 3	83.3 (s = 3.2)

Table 3.1: Average prototypes' SUS scores, refinement phase

score of 74.6, which has been described as 'good'. The prototype judged less usable by the interview participants was the second one, with an average score of 72.2, classified as in between 'ok' and 'good'.

3.3 Selection

The third and last step in our research methodology was the selection of one of the low-fidelity prototypes to carry on our work with a high-fidelity implementation.

For this purpose we designed an online survey, using the SurveyGizmo [59] platform, which we shared with developers in our network and online. The survey, which can be found in its entirety in A.2, was structured as follows:

- An introduction to our research, describing the goal of the survey, its structure and the estimated time to complete it.
- A series of demographic questions inquiring the participant's experience and usual practices with programming and code reviews.
- The evaluation of the prototypes. The prototypes were presented and evaluated one at time, in a random order, and the participant could decide to evaluate one, two or all the three of them. They were asked to watch a short (< 5 min) video walkthrough explaining the prototype's interface and the ways to interact with it. Then, they had to fill a SUS [17] questionnaire and could optionally provide comments about the prototype in a text-box.
- After the evaluations, the participant could provide general suggestions and remarks about the survey and the prototypes.

In total, 15 developers participated to the survey. Their background in professional software development was varied, ranging from none (students) to 5 years of experience. All the participants but one of them had at least 3 years of experience with programming, which the vast majority was doing daily. Most of them were doing code reviews at least once a week, mainly using the GitHub pull request interface. They were mostly satisfied with their code review environments, while the ones that reported dissatisfaction remarked as problematic the analysis of big pull requests ("Big pull requests are hard to grasp", "When the diff is large, it takes time to sort through relevant changes.") and the navigation of the changeset's context

(“Navigation between files is difficult [...] there is a lack of context”, “There are no IDE-like navigation features”).

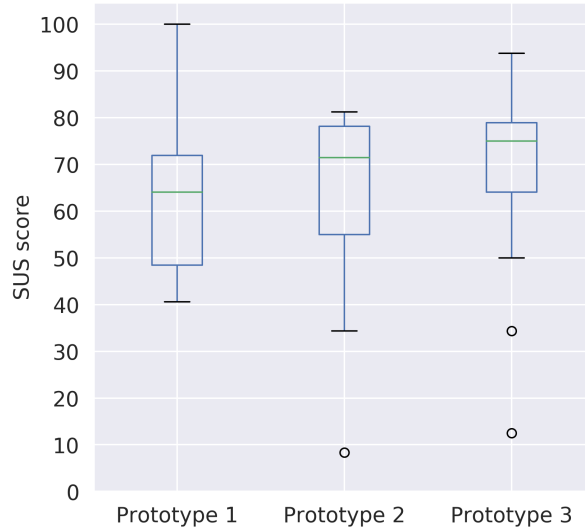


Figure 3.8: SUS scores, selection phase

In Figure 3.8 we present the distribution of SUS scores obtained by the prototypes. The best average score was obtained by the third prototype ($SUS_3^S = 75.0$), with the second ($SUS_2^S = 71.4$) and first ($SUS_1^S = 64.1$) prototypes following. To evaluate the significance of the difference in the obtained SUS scores’ means, we performed a t -test for each pair of prototypes, testing the hypothesis that the true means of the scores differ. In Table 3.2 we report the resulting p -values, which did not allow us to conclude that the means differ significantly.

	p -value
(P_1, P_2)	0.335
(P_1, P_3)	0.295
(P_2, P_3)	0.428

Table 3.2: p -values of the t -test of the prototypes’ SUS scores, in pairs

Next, we analysed the qualitative feedback. What emerged from the comments is that while the first two prototypes sparked interest (e.g., “Giving a visual representation of the changes can be interesting to understand at a first glance a lot of the characteristics of the change” [PR1]), they were also regarded as too complex and different from the code review environments commonly used by developers (e.g., “The prototype is too complex. When reviewing, what I would like to see is old code, new code. That’s it.” [PR1]). The need for the source code differences

3. PROTOTYPING PHASE

to be the primary focus was a common remark: “Usually I want to be focused on changes in code” [PR2].

Finding 1. Code review interfaces visually representing the structural changes spark interest, but are regarded as too complex. Developers want the source code differences to be the primary focus.

In agreement with this, the third prototype reported widespread appreciation (“I find this tool much easier than the first one [...] the idea of having callers and callees is quite nice”), despite a remark about the lack of useful information in comparison with the first two prototypes (“I think the other prototype shows more useful information”).

Finding 2. The navigation of the context of the changeset based on the callgraph is perceived as useful.

In accordance with the mean SUS score and the qualitative findings, we therefore decided to select the third prototype for a high-fidelity implementation. In the next chapter we will describe the implementation process of the tool, the design choices we made and the characteristics of its interface.

3.4 Threats to Validity

In this section, we present the threats to the validity of our research methodology.

Refinement. During the RITE phase, the four iterations of our prototypes were separately evaluated by two developers at time. It may be argued that such a small group size is not adequate to uncover all the existing usability issues in the interfaces. However, Lewis and Raton [60] claimed that, especially when the likelihood of problem occurrence is not small, the usage of small samples in a usability study is legitimate. Considering our iterative refinement process, in which we did not redesign the whole interface at each iteration but only fixed the problems found, we believe the sample size is acceptable.

Another threat to the validity stems from the way the study participants’ tested the prototypes. Given that they consisted in storyboards and were presented as a sequence of interactions, the participants could only evaluate the perceived usability of the interfaces, being unable to freely interact with a working version of them. To mitigate this threat we tried to make the storyboards as complete as possible, detailing all the available interactions and ensuring consistency in the way the three interfaces were presented. For the same reason, it was not possible to determine beforehand a set of tasks that the participants should have been able to complete to consider the prototype problem-free. The stop criteria was therefore manually established by the author, which may have introduced some bias in the process.

Selection. As for the refinement phase, a threat originates from the testing of the prototypes. In fact, the participants were asked to assess the (perceived) usability of the interfaces after watching a video walkthrough that described them, not being able to interact with a working prototype. Moreover, there was an inherent trade-off between a longer video that allowed the participants to get a clearer picture of the interface and its features, and a shorter video that did not discourage them to complete the survey. We tried to strike a balance between the two.

While the sample size of our selection survey was large enough to draw conclusions on the scores assigned to our prototypes, given that providing comments on the prototypes was optional, our qualitative findings are not supported by a sizeable number of developers, which may pose difficulties in generalising them.

Chapter 4

CHANGEVIZ: The Implementation

In this chapter we present the high-fidelity prototype review environment that we built, which we named CHANGEVIZ. First, we illustrate its interface and provide an overview of its features. Next, we briefly present the infrastructure behind it. We conclude discussing the limitations of our tool and pointing out directions for future work.

4.1 Interface

CHANGEVIZ is an open-source tool¹ designed to perform the code review of pull requests to any Java project on GitHub. In Figure 4.1 we display its interface, visualising pull request #2789 of Retrofit [1]. The interface is divided into two main parts: the pull request URL input area at the top and the changeset visualisation at the bottom. To perform a review, the user inputs the pull request URL and clicks on the 'Load' button.

The changeset visualisation resembles the one proposed by GitHub and other popular code review environments. The unified diffs of each modified file are shown one after the other and syntax highlighting is applied to the code, emphasising the language keywords. To expand a diff's context, the user can click on the path of a modified file at the top of it. A modal window showing the full diff (infinite context) will appear on the screen, scrolled automatically to the starting position of the diff portion initially shown.

The main difference between CHANGEVIZ and the GitHub review interface is the presence of two sidebars at the sides of each diff. The sidebars are designed to allow the reviewer to navigate the context of the changeset following the method call relationships.

The *callers* sidebar on the left (Figure 4.2) resembles the 'Find References' feature commonly offered by IDEs. For each method whose declaration is (partly, or fully) contained in the portion of code shown in the diff, it displays automatically the list

¹<https://github.com/joined/TUdelft-VisualisationOfCodeChanges>

4. CHANGEVIZ: THE IMPLEMENTATION

Visualization of Code Changes for Code Review

A MSc thesis project by Lorenzo Gasparini (gasparini.lorenzo[at]gmail.com).

Please insert the URL of the pull request which you want to perform the review on, and click the load button (or press enter):

The screenshot displays the CHANGEVIZ interface for a pull request at `https://github.com/square/retrofit/pull/2789`. It shows a diff for `retrofit/src/main/java/retrofit2/Retrotif.java` and `retrofit/src/main/java/retrofit2/ServiceMethod.java`. The interface is divided into two main sections: 'CALLERS' and 'CALLEES'. The 'CALLERS' section on the left lists references to methods in the diff, including file names, line numbers, and the method call itself. The 'CALLEES' section on the right lists the definitions of those methods, also with file names, line numbers, and the method definition. The code is syntax-highlighted and includes line numbers and change indicators (e.g., @@ -143,10 +143,7 @@). The interface also includes a 'Load' button and a URL input field.

Figure 4.1: CHANGEVIZ interface

of references (i.e., calls) to the method in other parts of the code base. More specifically, it displays the following information about each reference to the method: the name of the Java file containing the method call, its location in terms of line number, and the method call itself. If the file containing the reference was modified by the changeset currently being reviewed, a pencil next to the filename indicates it. Each list of references is vertically aligned with the corresponding (portion of) method declaration, establishing a visual connection between the two, and a scrollbar can be used to browse it if there is a large number of references. When a specific method call is hovered with the mouse cursor, an information box appears (Figure 4.2b), providing the path of the file containing the reference and the complete method call. By clicking on the method call, moreover, the user can access the source code of the file that contains it, which is displayed in a modal window (Figure 4.4). The modal window content varies depending on whether the file in question was modified or not: if the file was modified, it shows the unified diff of the file, otherwise it shows its source code. In both cases, the code is syntax-highlighted and automatically scrolled to the position of the method call, whose line number is highlighted to ease its location. A click outside of the modal window boundaries closes it.

The *callees* sidebar on the right (Figure 4.3) mimics the ‘Go to Definition’ feature that IDEs offer. It allows the reviewer to obtain information about the definition of the methods that are invoked within the diff (callees), excluding the ones declared

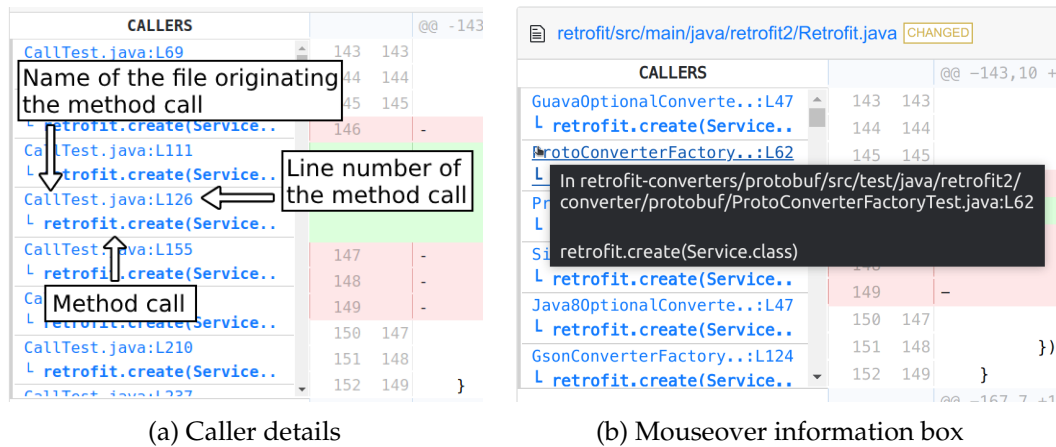


Figure 4.2: Callers sidebar

in external libraries. For each method call in the diff, the sidebar displays the following information: name of the file containing the called method's definition, line range of the definition within that file, and the signature of the called method. As for the callers sidebar, a pencil next to the filename indicates whether the file containing the method definition was affected by the changeset in analysis. The callee information is vertically positioned on the same line as the corresponding method invocation, which is enlarged when it contains multiple method calls. By hovering a callee, the user can obtain additional details about it, namely: the path of the file containing the method definition and the qualified signature (i.e., including the package and the class it belongs to) of the method. As for the callers, clicking on a callee opens the source code (or diff, if it was modified) of the file that contains its definition. The code is scrolled to the start of the method declaration, and the

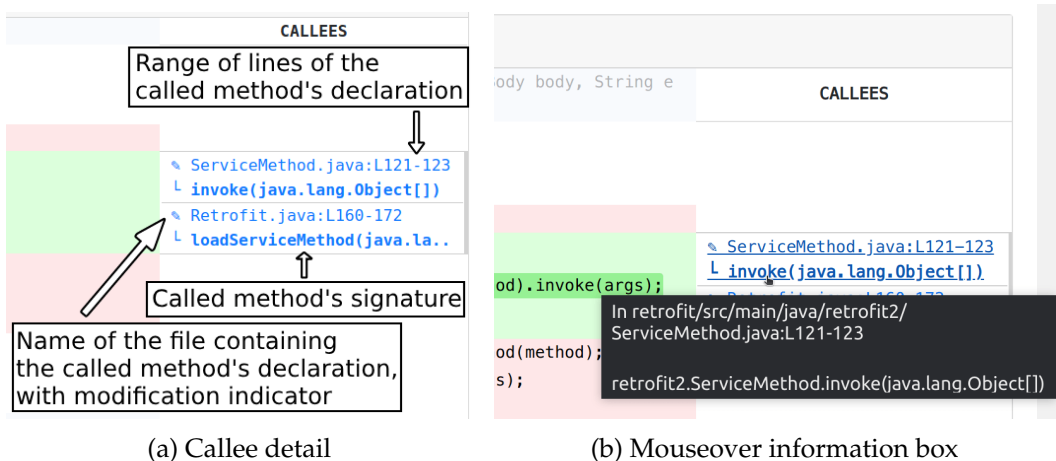


Figure 4.3: Callees sidebar

4. CHANGEVIZ: THE IMPLEMENTATION

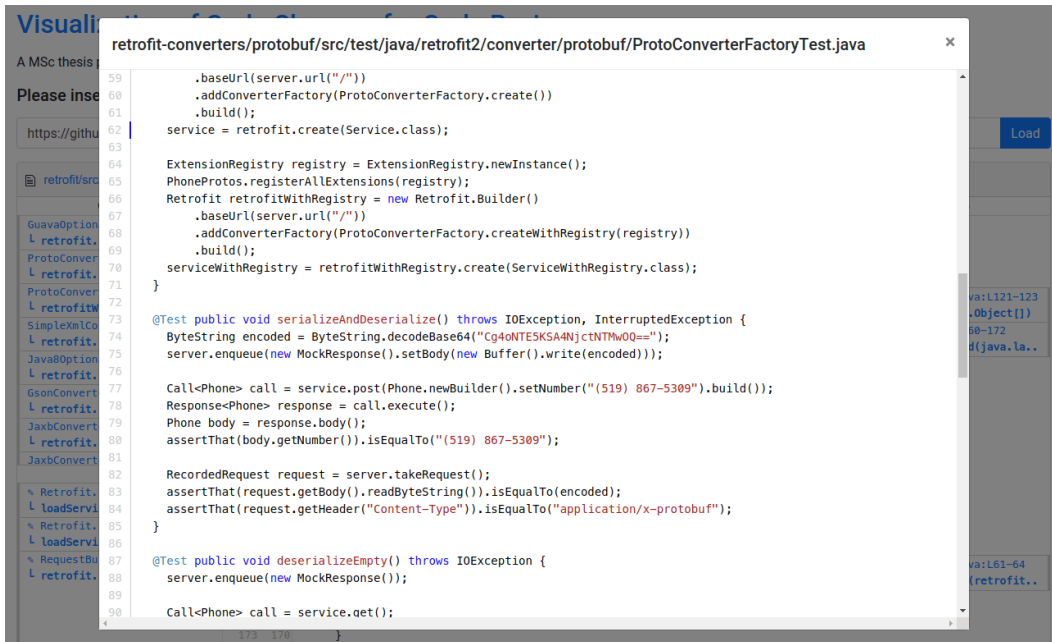


Figure 4.4: Caller display modal window

corresponding line range is highlighted.

4.2 Infrastructure

CHANGEVIZ is built as a web-based tool, making it possible for any developer to access it using a web browser and perform the code review of pull requests to Java projects on GitHub without installing additional software. For the implementation of the tool, we leveraged web technologies and realised an infrastructure (Figure 4.5) with 3 interconnected parts: the front-end, the back-end, and the method call extraction engine.

The front-end refers to the web application that is executed in the user's browser, consisting of a mixture of HTML, CSS and JavaScript files, that manages the presentation and layout of the data. As soon as the user inputs a valid pull request URL and clicks on the 'Load' button (or presses enter), the application sends two requests: one to the GitHub API to obtain the pull request's diff and another one to the back-end to obtain the callers and callees information. The diff is immediately visualised while the application awaits a response from the back-end containing the method call information.

The back-end, which manages the state of the tool, is a Python program running on a server that uses the Flask web framework to expose an HTTP API to communicate with the front-end. As computing the callers and callees information is a resource- and time-intensive task, especially for large code bases, an SQLite

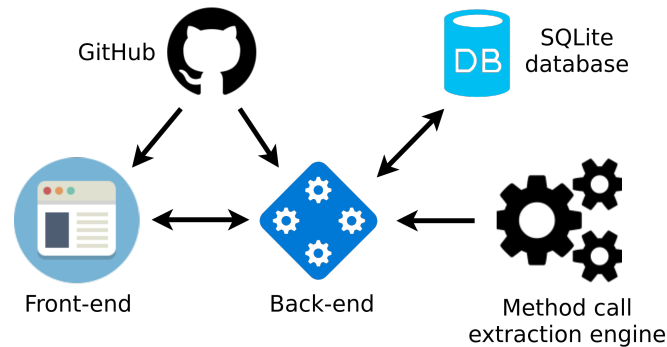


Figure 4.5: Infrastructure overview

database is used to cache the computed method call information. In order to compute the callers and callees of a pull request, the back-end retrieves its diff from GitHub and clones the entire Git repository locally. The diff is then applied to obtain the state of the code base after the application of the pull request, and the method call extraction engine is invoked to extract the method call information of each modified Java file. The method call extraction engine is a Java program that uses the `JavaParser` and `JavaSymbolSolver` libraries to perform the static analysis of a Java code base and extract the callers and callees of a given set of files. When the extraction is completed, the computed method calls are cached in the database and sent to the front-end which visualises them in the sidebars.

When the user clicks on a modified file to expand its context, a caller or a callee, the front-end sends a request to the back-end to obtain the source code of the given file. The back-end restores the state of the Git repository to the one that follows the application of the pull request, and extracts either the source code or the full diff (infinite context) of the requested file, which is then sent to the front-end that visualises it in the modal window.

4.3 Discussion

During the conversion of the low-fidelity prototype to the high-fidelity implementation, a number of choices were made in order to make the realisation of the tool possible.

The callers and callees information of the modified files is computed on their newer version (i.e., after the application of the pull request). This was done to avoid the confusion that would have resulted from the overlap of the method call information of the old and new version of the files. Although we argue that the new version of the files is the one of which the reviewers are the most interested in exploring the context, there may be an added value in allowing them browse the context of the old version of the files as well.

When a method call in the diff references a method which is declared in an external library, inspecting its definition is not possible. This limitation stems from the

4. CHANGEVIZ: THE IMPLEMENTATION

variety of build systems used by Java projects, that complicates the automated retrieval and resolution of external libraries. Given the widespread usage of libraries, the possibility to inspect their references might be a useful addition.

In the low-fidelity prototype, the user was able to iteratively explore the context of callers and callees. During the development of CHANGEVIZ, this feature was abandoned. The results obtained by Kramer et al. [56] suggest, in fact, that navigating along multiple edges of the callgraph at once is too cognitively challenging.

We realised our tool as a standalone web application, that developers need to visit to review their GitHub pull requests with the context-exploration features that CHANGEVIZ offers. A future iteration could be offered as a browser extension that integrates with the GitHub review environment, which most developers are widely accustomed with.

Chapter 5

Evaluation

In this chapter we present a preliminary evaluation of CHANGEVIZ. The goal is to assess its effectiveness both from quantitative and qualitative perspectives, as well as to collect developers' feedback and point to further improvement directions for tool-based reviewer support.

First, we describe the method that we used for the assessment. We continue presenting the results of the experiments, and finish with a discussion of the outcomes along with the threats to the validity of our experimental setup.

5.1 Methodology

The evaluation is centred on the following research question:

RQ: Does CHANGEVIZ effectively support the reviewer when performing code reviews?

To explore its effectiveness, we designed an experiment involving 10 developers performing two code review tasks, comparing our tool to the GitHub pull request interface. Each of these two tasks consisted in the time-limited review of a pull request to a Java software code base, unknown to the participants, containing a defect. The task was considered to be solved when the participants identified the defect and described it.

In particular, we tested the following hypotheses:

- **H1:** The probability of solving a time-limited task involving the review of a pull request to an unknown code base using CHANGEVIZ is higher than the probability of solving the same task using the GitHub pull request interface.
- **H2:** Programmers can solve a task involving the review of a pull request to an unknown code base quicker using CHANGEVIZ than using the GitHub pull request interface.

5. EVALUATION

```
src/main/java/org/jabref/gui/util/FileDialogConfiguration.java [CHANGED]
CALLERS
61 61 @@ -61,6 +61,9 @@ public void setSelectedExtensionFilter(FileChooser.ExtensionFilter selectedExten
62 62     private String initialFileName;
63 63     public FileDialogConfiguration build() {
64 +     if (initialFileName == null) {
65 +         initialFileName = "ExportedReferences.html";
66 +     }
64 67     return new FileDialogConfiguration(initialDirectory, extensionFilters, defaultExtension, initialFileName);
65 68     }
66 69 }
```

Figure 5.1: Changes to ‘FileDialogConfiguration.java’ in task 2, in CHANGEVIZ

The group of participants was divided in two: the *control group* using the standard GitHub interface to perform the review tasks and the *treatment group*, which was instead required to use our own code review environment. The candidates were equally divided into the two groups. The surveys presented to each group can be found in their entirety in A.3 and A.4, respectively.

In this context we define a defect as *external* if it exists because of the propagation of the side-effects of the changes outside of the changeset due to callgraph or class hierarchy relationships, as opposed to an *internal* defect which lies directly within the changeset and, therefore, can be noticed without navigating its context.

First, participants were asked a number of demographic questions about their usual programming and code review practices. Then, participants of the treatment group were allowed 5 minutes to familiarise themselves with the interface of CHANGEVIZ on an unrelated pull request. Finally, participants of both groups were asked to perform the following code review tasks:

- **Task 1:** The review of a pull request containing an *internal* defect, which should supposedly lead to similar success rates and completion times for both groups.

For this task a time limit of 10 minutes was given.

- **Task 2:** The review of a pull request containing an *external* defect, which should highlight the capabilities of our tool and therefore lead to a higher success rate and shorter task completion times for the treatment group.

For this task a time limit of 25 minutes was given, and the participants were given a hint after 15 minutes from the start, had they not yet found the defect in the code. The hint was the following: *think about the side-effects of the changes in ‘FileDialogConfiguration.java’.*

In fact the defect consists in the unwanted propagation of side-effects due to the changes to the method ‘build()’ of this file, which, as shown in Figure 5.1, has a number of callers which are affected by the change, although they are not related to the intent of the pull request.

When participants of the treatment group did not manage to find the defect within the given time limit, we stopped the experiment and asked what would have allowed them to discover it, and collected their comments.

The pull requests are modifications to the version 4.3.1 of the JabRef¹ code base, a Java reference management software. We chose this project because it is reasonably complex (120.000 lines of code) and is actively developed on GitHub, supporting the idea of using CHANGEVIZ as an alternative to the standard GitHub interface.

For each task, a description of the pull request was given, explaining its rationale. Participants were also requested not to focus on code style and syntax issues, as the defects had a functional nature (i.e., they affected the functionality of the application).

The hardware setup for the experiment was a 15" laptop with a 1920x1080 pixel screen resolution, an Intel i7-7700HQ CPU and 16GB of RAM. The review environments were accessed using the Google Chrome browser version 69 running on the Ubuntu 17.10 operating system.

5.2 Quantitative Results

Out of all the 10 participants, 6 were postgraduate students, 1 was a PhD student, and 3 were professional software developers. All the interviewees were familiar with the GitHub pull request review interface, and none of them was familiar with JabRef. Only two of the students had professional programming experience, but most of them had at least 2 years of Java programming experience. The majority of the interviewees programmed at least once a day, and all but two of them performed code reviews at least once a month. The complete results can be found in Appendix A.5.

5.2.1 Task 1: Internal Defect

All the participants of both groups were able to find the defect in the first task. In terms of review success rate, therefore, we have no evidence that CHANGEVIZ performs in a worse way than the GitHub interface, for an internal defect.

The candidates of the control group had an average completion time of 5m36s (STD = 1m47s) while the ones in the treatment group had an average completion time of 4m42s (STD = 1m21s), as can be seen in Figure 5.2.

The *t*-test does not lead to the rejection of the null hypothesis of equal population means for the two groups ($t = 0.90$, $p = 0.39$). The normality assumption of the *t*-test is challenged, given the reduced sample size, but the Shapiro-Wilk normality test does not result in the rejection of the null hypothesis of a normally distributed population for each group at a $p = 0.05$ significance level, suggesting that the *t*-test is still appropriate. The results therefore support our conjecture that the completion time does not change significantly between the two interfaces for this type of task.

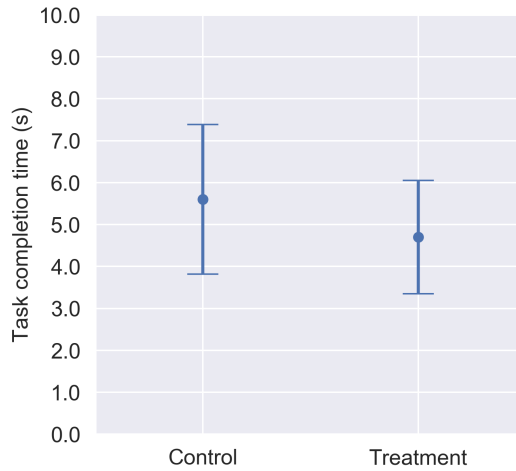


Figure 5.2: Task 1 completion times, per group

	Control group	Treatment group
Defect found	0	2
Defect not found	6	4

Table 5.1: Task 2 results, per group

5.2.2 Task 2: External Defect

Table 5.1 displays the results for the second task. None of the candidates using the GitHub interface to perform the review were able to find the defect, while only two candidates who used our tool managed to find it. Fisher’s exact test for the contingency table yields $p = 0.45$, failing to detect a significant association between the groups and the task outcomes.

The two candidates who successfully completed the task, both professional developers with more than 5 years of Java programming experience, did so in 13m30s and 23m respectively, with the latter being given the hint after 15 minutes.

5.3 Qualitative Results and Discussion

The results for the first task fit with our expectations, showing equal success rates and (variations due to the reduced sample size aside) completion times for both the groups using the GitHub interface and CHANGEVIZ. This supports our hypothesis of a feature parity of the two interfaces for this type of task. Even though for the first task the usage of the sidebars provided by our tool was not necessary to find the defect (as it was in the changeset itself), most participants of the treatment

¹<http://www.jabref.org/>

group tried immediately to use them, thinking that they would have been crucial to complete this task. After a brief experimentation with the sidebars to navigate the context and get an overview of it, though, they did not seem to find any interesting portion of code to analyse in detail, and therefore quickly went back to focusing on the main changeset as they would have normally done with the GitHub interface. As already noted, this initial context exploration did not negatively affect the average completion time, if compared to the control group. We suppose that this behaviour was caused by the fact that developers were new to CHANGEVIZ, and being used to the GitHub pull request view, they felt the urge to make use of the navigation features offered by our interface, which they just learned about during the 5 minutes introduction to the tool.

In the second task, no participant of the control group was able to find the defect. As GitHub's review interface does not provide any support to navigate the context of the changed code, they focused only on the changeset initially, and no one tried to explore other classes in the code base. When they were given the hint after 15 minutes, some of them tried to understand how the changes to 'FileDialogConfiguration.java' and its 'build()' method affected other classes in the code base, but they quickly understood that this is a difficult task to perform on GitHub. Not having any 'Find References'-like feature available, they tried to make use of the search functionality to identify the usages of the 'FileDialogConfiguration' class, discovering that GitHub does not offer a full-text search feature for the code base of a repository at a specific commit. Realising this was not an option, they went back to the changeset and tried again to identify the defect therein, with no success.

On the other hand, two developers of the treatment group were able to find the defect. These were professional developers with significant experience in professional software development (3 to 5 and 6 to 10 years, respectively) and Java (6 to 10 years). Their extensive programming experience (compared to the other participants) may have allowed them to understand more quickly the structure of the modified classes and therefore reason about the code in more high-level terms. Most of the other developers, especially the ones with less Java programming experience, initially spent a significant amount of time trying to make sense out of the way that the code was written, being less familiar with the language and its design patterns, including the 'Builder' pattern (used within the modified files). A professional developer, with a similar amount of experience as the two participants who successfully completed the second task, was part of the control group; despite his familiarity with the language, he did not find the defect. This supports the hypothesis that the features offered by our tool are effective in performing this type of task.

We asked the developers of the treatment group who failed the second task what would have helped them to notice it. All of them, after being given the detailed description of the defect, remarked that it was reasonably easy to spot using our tool's features, being astonished not to have found it during the experiment. Surprisingly, most of them did not make a significant use of the left sidebar to anal-

use the side-effects of the changes, even after being told to focus on that aspect with the 15-minutes-mark hint. Based on their navigation behaviour during the experiment and their feedback, we suppose that developers got overwhelmed by the amount of information displayed in a way they were not used to, not having had enough time to get accustomed to the tool's novel layout. We conjecture that this problem could be mitigated once developers gain experience with CHANGEVIZ and make it part of their workflow. Among the common remarks there was the request for a better visual connection between the callers and callees in the sidebars and the method definitions and method calls in the changeset, respectively. This could also encourage developers to make more use of the sidebars' features. A developer suggested to make each method call in the changeset clickable, replicating the behaviour of the click on a callee in the sidebar, which opens the class containing the called method definition and highlights it. Another developer remarked that a separation between the test and non-test classes in the callers sidebar would be ideal, displaying first the non-test ones; the rationale in this case is that unwanted side-effects in the test files are less important, as they should already be highlighted by the execution of the test suite.

5.3.1 Threats to Validity

In this section, we present the threats to the validity of our experiments.

Subjects. The reduced sample size threatens the results of our experiment. We tried to mitigate this limitation by including participants with a wide range of programming backgrounds, in an attempt to improve the representativeness of the subjects and the generalisability of our results.

Tasks. We created both the pull requests that the participants had to review in their tasks, which may have been to the advantage of our tool. We mitigated this by describing the rationale of the pull requests and making them available online, through the links in A.3 and A.4. Given the large number of participants who did not manage to solve the second task, another threat is possibly represented by a too short time given to complete it. The time limit, though, was decided based on the outcome of a pre-test, and no participant made any remark about needing more time when we asked them what would have helped them to solve the task, at the end of the experiment.

Training. Only the participants of the treatment group received an initial 5-minutes training, since we made sure that everyone was already familiar with the GitHub pull request interface. Looking at the task results, a longer training could have been helpful. However long the initial training would have been, though, it could not have gotten the participants as familiar with CHANGEVIZ as they were with the GitHub interface, given that they used the latter for a vastly larger period of time.

Chapter 6

Conclusions

In this chapter we conclude our research, providing an overview of our contributions, highlighting the limitations of our approach and pointing out directions for future work.

6.1 Contributions and Limitations

Our main contribution is *CHANGEVIZ*, a tool to perform code reviews that provides cognitive support to reviewers, allowing them to explore the context of a changeset based on the callgraph. The tool was developed on the basis of a survey of the tool-based cognitive support approaches that we presented in Chapter 2. Based on the survey results, we built a set of candidate interfaces which we refined interviewing their prospective users, and selected one of the prototypes through an online survey for a high-fidelity implementation. In Chapter 4 we described *CHANGEVIZ*, our web-based review environment that supports the review of any pull request to Java projects on GitHub.

The second contribution consists of a preliminary evaluation of the effectiveness of the tool that we developed, provided in Chapter 5. We evaluated *CHANGEVIZ* through an experiment in which developers were asked to perform two code review tasks using our tool and the GitHub pull request interface, a common choice of review environment. Through the experiment we collected both quantitative and qualitative feedback of *CHANGEVIZ*, which helped identify its key strengths and shortcomings.

Among the limitations of our research, there is the restricted population of developers that took part in the experiments. They were, for the most part, postgraduate students in Computer Science that did not have multiple years of professional experience with programming and code reviewing, threatening the generalisability of our results. Moreover, our implementation presents some technical limitations related to features that were deemed out of scope within our research. Although the interface could, in principle, suit the visualisation of code changes in any object-oriented programming language, the initial implementation of *CHANGEVIZ* sup-

ports only the review of Java changesets. As our goal was to provide support in the understanding of changes rather than building a fully-fledged review environment, the tool lacks some of the features commonly found in a review tool, such as the ability to add a comment on a specific range of lines.

6.2 Conclusions and Future Work

We made a contribution to the field of cognitive support review tools and evaluated its effectiveness in a research experiment, which we enable researchers to replicate by sharing its related material¹. The results, albeit difficult to generalise due to the small sample size, indicate that the context navigation features offered by CHANGEVIZ are deemed useful and can effectively help reviewers to understand a changeset. Being this a key challenge in the code review process, we conjecture that these features could be a valid help for reviewers.

A thorough evaluation of our tool in a real-world usage scenario is still needed, in order to pinpoint the validity of our approach. Involving a large group of developers with multiple years of experience in professional software development could provide additional insights into the goodness of CHANGEVIZ.

Future research could also investigate the integration of the context navigation features that we proposed into one of the existing code review environments, evaluating whether the effectiveness of the code exploration is hindered or promoted. CHANGEVIZ could be proposed as a browser extension that transparently enhances the GitHub pull request view, removing the friction caused by the navigation to a different website, and thus increasing the chances of adoption of the tool by reviewers.

¹https://github.com/joined/TUdelft-VisualisationOfCodeChanges/tree/master/Related_Material

Bibliography

- [1] Square Retrofit pull request #2789 on GitHub. <https://github.com/square/retrofit/pull/2789>. [Online].
- [2] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. A Faceted Classification Scheme for Change-Based Industrial Code Review Processes. In *Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, pages 74–85. IEEE, 8 2016.
- [3] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: The integrator’s perspective. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 358–368. IEEE, 5 2015.
- [4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings - International Conference on Software Engineering*, pages 712–721, 2013.
- [5] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 192–201, New York, New York, USA, 2014. ACM Press.
- [6] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 202, 2013.
- [7] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software*, 35(4):34–42, 7 2018.
- [8] Tobias Baum and Kurt Schneider. On the Need for a New Generation of Code Review Tools. In *Product-Focused Software Process Improvement*, pages 301–308. Springer, Cham, 11 2016.

- [9] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 134–144. IEEE, 5 2015.
- [10] Luis Fernando Cortes-Coy, Mario Linares-Vasquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*, pages 275–284. IEEE, 9 2014.
- [11] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 111–122. IEEE, 5 2015.
- [12] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D’Hondt. Visually characterizing source code changes. *Science of Computer Programming*, 98(P3):376–393, 2 2015.
- [13] Marco D’Ambros, Michele Lanza, and Romain Robbes. Commit 2.0. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering - Web2SE ’10*, pages 14–19, New York, New York, USA, 2010. ACM Press.
- [14] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. Code review: Veni, ViDI, vici. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 151–160. IEEE, 3 2015.
- [15] Michael C. Medlock, Dennis Wixon, Mark Terrano, Ramon L. Romero, and Bill Fulton. Using the RITE method to improve products; a definition and a case study. *Usability Professionals Association*, 2002.
- [16] Anker Helms Jørgensen. Thinking-aloud in user interface design: A method promoting cognitive ergonomics. *Ergonomics*, 33(4):501–507, 4 1990.
- [17] John Brooke. SUS - A quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [18] GitHub. <https://github.com/>. [Online].
- [19] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [20] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP ’18*, pages 181–190, New York, New York, USA, 2018. ACM Press.

-
- [21] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken Ichi Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 141–150. IEEE, 3 2015.
- [22] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings - International Conference on Software Engineering*, pages 931–940. IEEE, 5 2013.
- [23] J W Hunt and M D Mcilroy. An Algorithm for Differential File Comparison. *Time*, 41(41):1–9, 1976.
- [24] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA, 2012. ACM Press.
- [25] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 33, New York, New York, USA, 2010. ACM Press.
- [26] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic Generation of Release Notes. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484 – 495, 2014.
- [27] Siyuan Jiang and Collin McMillan. Towards Automatic Generation of Short Summaries of Commits. In *IEEE International Conference on Program Comprehension*, pages 320–323. IEEE, 5 2017.
- [28] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. The effects of change-decomposition on code review - A Controlled Experiment. In *Proceedings of 12th International Symposium on Empirical Software Engineering and Measurement*, 5 2018.
- [29] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130. IEEE, 5 2013.
- [30] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *IEEE International Working Conference on Mining Software Repositories*, volume 2015-Augus, pages 180–190, 2015.

- [31] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stephane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 341–350, 2 2015.
- [32] Bo Guo and Myoungkyu Song. Interactively Decomposing Composite Changes to Support Code Review and Regression Testing. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 118–127. IEEE, 7 2017.
- [33] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings - International Conference on Software Engineering*, pages 309–319. IEEE, 2009.
- [34] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, 1 2013.
- [35] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *IEEE International Conference on Software Maintenance, ICSM*, pages 1–10. IEEE, 9 2010.
- [36] David Kawrykow and Martin P Robillard. Non-essential changes in version histories. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 351, 2011.
- [37] Sirinut Thangthumachit, Shinpei Hayashi, and Motoshi Saeki. Understanding source code differences by separating refactoring effects. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 339–347. IEEE, 12 2011.
- [38] Shinpei Hayashi, Sirinut Thangthumachit, and Motoshi Saeki. REdiffs: Refactoring-Aware difference viewer for java. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 487–488. IEEE, 10 2013.
- [39] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Václav Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proceedings - IEEE Workshop on Program Comprehension*, pages 149–152. IEEE, 2005.
- [40] Barbara G. Ryder, M. Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for Java programs. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 39(10):664–665, 2005.
- [41] Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 2, page 163, New York, New York, USA, 2010. ACM Press.

- [42] Matthew Follett and Orland Hoerber. ImpactViz: visualizing class dependencies and the impact of changes in software revisions. In *Proceedings of the 5th international symposium on Software visualization*, page 209–210, New York, New York, USA, 2010. ACM Press.
- [43] Fernanda M Delfim, Lilian P Scatalon, Jorge M Prates, and Rogerio E Garcia. Visual Approach for Change Impact Analysis: A Controlled Experiment. *2015 12th International Conference on Information Technology - New Generations*, pages 391–396, 2015.
- [44] Renato L. Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 11 2013.
- [45] Zhenchang Xing and Eleni Stroulia. Bottom-up design evolution concern discovery and analysis. Technical report, University of Alberta, 2007.
- [46] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, 2005.
- [47] Andrew McNair, Daniel M. German, and Jens Weber-Jahnke. Visualizing Software Architecture Evolution Using Change-Sets. *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.
- [48] Ronie Salgado and Alexandre Bergel. Pharo Git Thermite A Visual Tool for Deciding to Weld a Pull Request. In *Proceedings of the International Workshop on Smalltalk Technologies*, volume 610, Maribor, Slovenia, 2017. ACM press.
- [49] Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2006*, pages 187–194. IEEE, 2006.
- [50] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, page 2503, New York, New York, USA, 2010. ACM Press.
- [51] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and B Jan. Stackplorer: Call graph navigation helps increasing code maintenance efficiency. In *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 217–224, New York, New York, USA, 2011. ACM Press.

- [52] Seonah Lee, Sungwon Kang, and Matt Staats. NavClus: A graphical recommender for assisting code exploration. In *Proceedings - International Conference on Software Engineering*, pages 1315–1318. IEEE, 5 2013.
- [53] Mockito framework pull request #1247 on GitHub. <https://github.com/mockito/mockito/pull/1247>. [Online].
- [54] Google Slides. <https://www.google.com/slides/about/>. [Online].
- [55] Steven P. Reiss and Alexander Tarvo. Tool demonstration: The visualizations of code bubbles. In *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, pages 1–4. IEEE, 9 2013.
- [56] Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How tools in IDEs shape developers’ navigation behavior. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, page 3073, 2013.
- [57] Daniel R. Bradley and Ian J. Hayes. Visuocode: A software development environment that supports spatial navigation and composition. In *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, pages 1–4. IEEE, 9 2013.
- [58] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.
- [59] SurveyGizmo. <https://www.surveygizmo.com/>. [Online].
- [60] James R Lewis and Boca Raton. Legitimate Use of Small Samples in Usability Studies : Three Examples. (December), 1991.

Appendix A

Surveys and Results

A.1 Low-fidelity Prototype Evaluation: SUS

For each of the following statements, mark one box that best describes your reactions to the prototype:

	Strongly Disagree	Disagree	Neither	Agree	Strongly Agree
1. I think I would like to use this tool frequently.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2. I found this tool unnecessarily complex.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3. I thought this tool was easy to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4. I think that I would need assistance to be able to use this tool.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5. I found the various functions in this tool were well integrated.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6. I thought there was too much inconsistency in this tool.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7. I would imagine that most people would learn to use this tool very quickly.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8. I found this tool very cumbersome/awkward to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please provide any comments about this prototype: _____

A.2 High-fidelity Implementation Prototype Selection

Welcome to the survey for the evaluation of the “Visualisation of Code Changes for Code Review” prototypes.

My name is **Lorenzo Gasparini** and I am a researcher in Software Engineering at TU Delft. In this research, together with Alberto Bacchelli (ZEST, University of Zurich) and Tobias Baum (Leibniz Universität Hannover), we are focusing on improving the code review process using software visualisation.

Our goal is to help reviewers in their review process by supporting them with visualisations of the changes to be reviewed.

We have devised three visualisation prototypes and created this survey to collect your opinion before proceeding with the actual implementation of one of these prototypes. Our research is currently focused on the **Java** programming language, therefore terms such as ‘package’, ‘class’ and ‘method’ are to be interpreted accordingly.

All of the prototypes that you are going to evaluate are visualising the same set of changes, in particular the pull request #1247 of the **Mockito** open-source Java software. This gives you an idea of the information that we are trying to visualise.

The survey is structured as follows:

- Some demographic questions about your usual programming and code review practices.
- The evaluation of the prototypes:
 - The evaluation consists of a video explaining the prototype, a questionnaire about your opinion on the prototype, and a suggestions/remarks text box.
 - At the end of the prototype evaluation, you decide if you want to evaluate another prototype (up to three) or end the survey.
- A general remarks/suggestions text box for the survey and the prototypes.

The survey will take approximately 10-20 minutes to complete, basing on how many prototypes you decide to evaluate. All the data that you provide will be used in aggregated and anonymized form and for research purposes only; the data may be shared with the research community to allow replication and verifiability of our results.

By pressing Next, you agree that your data will be stored and used in an anonymized form. If you have any questions/remarks/suggestions about the survey, please send an email to l.gasparini@student.tudelft.nl.

A.2. High-fidelity Implementation Prototype Selection

1. How long have you been doing the following activities?

	Never done it, so far	1 year or less	2 years	3-5 years	6-10 years	11 years or more
Professional software development	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

2. How often do you currently perform the following activities?

	Never	Yearly	Monthly	Weekly	Daily or more often
Programming	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Code reviewing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

3. What setting do you work in?

- Open source
- Industry, proprietary

4. Which company do you work for?

5. Which open source project(s) do you work on?

6. Which of the following best describes your primary work area?

- Development
- Test
- Visual Design
- Documentation
- Product Support
- Management and Administration
- Research
- Other: _____

7. How many developers are in your immediate team?

A. SURVEYS AND RESULTS

8. Which code review tool(s)/environment(s) do you use?

- Gerrit
- Atlassian Crucible
- Atlassian Bitbucket Server (formerly Stash)
- JetBrains Upsource
- SmartBear Collaborator
- ReviewClipse
- Microsoft CodeFlow
- Github Pull Requests
- Phabricator
- GitLab
- General software development tools (IDE, SCM, ticket system / bug tracker)
- Other: _____

9. How satisfied are you with your code review environment?

- Completely dissatisfied
- Mostly dissatisfied
- Somewhat dissatisfied
- Neither satisfied or dissatisfied
- Somewhat satisfied
- Mostly satisfied
- Completely satisfied

10. What are the main reasons why you are not satisfied with your code review tool?

11. Prototype 1 evaluation:

Prototype 1 Video Walkthrough ▷

a) After watching the video, please fill in the usability questionnaire:

A.2. High-fidelity Implementation Prototype Selection

		Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	I don't know
1.	I think I would like to use this tool frequently.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
2.	I found this tool unnecessarily complex.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
3.	I thought this tool was easy to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
4.	I think that I would need assistance to be able to use this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
5.	I found the various functions in this tool were well integrated.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
6.	I thought there was too much inconsistency in this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
7.	I would imagine that most people would learn to use this tool very quickly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
8.	I found this tool very cumbersome/awkward to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

b) Do you have any remarks/suggestions on this prototype?

12. Prototype 2 evaluation:

Prototype 2 Video Walkthrough ▷

a) After watching the video, please fill in the usability questionnaire:

		Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	I don't know
1.	I think I would like to use this tool frequently.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
2.	I found this tool unnecessarily complex.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
3.	I thought this tool was easy to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
4.	I think that I would need assistance to be able to use this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
5.	I found the various functions in this tool were well integrated.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
6.	I thought there was too much inconsistency in this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
7.	I would imagine that most people would learn to use this tool very quickly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
8.	I found this tool very cumbersome/awkward to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

A. SURVEYS AND RESULTS

b) Do you have any remarks/suggestions on this prototype?

13. Prototype 3 evaluation:

Prototype 3 Video Walkthrough ▷

a) After watching the video, please fill in the usability questionnaire:

	Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	I don't know
1. I think I would like to use this tool frequently.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
2. I found this tool unnecessarily complex.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
3. I thought this tool was easy to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
4. I think that I would need assistance to be able to use this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
5. I found the various functions in this tool were well integrated.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
6. I thought there was too much inconsistency in this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
7. I would imagine that most people would learn to use this tool very quickly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
8. I found this tool very cumbersome/awkward to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

b) Do you have any remarks/suggestions on this prototype?

14. If you have any comments about the survey and/or the prototypes, please type them here.

Thank you for taking our survey. Your input will be used to improve the state of the art in code reviewing support tools!

In case you are interested in other parts of our research on code reviews, you can have a look at our group's website: <http://www.ifi.uzh.ch/en/zest.html>

Feel free to contact me at l.gasparini@student.tudelft.nl for any question you may have about the research.

A.3 Tool Evaluation: Control Group

Welcome to this survey on the evaluation of my master's research project.

My name is Lorenzo Gasparini and I am a MSc student in Computer Science at TU Delft. For my master's thesis, supervised by Prof. Alberto Bacchelli (ZEST, University of Zurich) and M. Eng. Tobias Baum (Leibniz Universität Hannover), we are performing a research on code reviews. Your help is appreciated.

A.3.1 General questions

Please start by answering the following questions:

1. What is your main role?
 - Programmer / Software Developer
 - Tester
 - Analyst
 - Software Architect
 - IT Operations
 - Manager
 - Researcher / Professor / PhD Student
 - Student
 - Other
2. How many years ago did you start professional software development?
 - I have no experience in professional software development
 - 1 year or less
 - 2 years
 - 3 - 5 years
 - 6 - 10 years
 - 11 years or more
3. How many years ago did you start programming in Java?
 - I have never programmed in Java so far
 - 1 year or less
 - 2 years
 - 3 - 5 years
 - 6 - 10 years
 - 11 years or more

4. How often do you currently do programming?

- Not at all
- About once a year
- About once a month
- About once a week
- About once a day or more often

5. How often do you currently perform code reviews?

- Not at all
- About once a year
- About once a month
- About once a week
- About once a day or more often

A.3.2 Code review

Finally, you are assigned two code review tasks. They consist in the review of two pull requests to the JabRef project, a bibliography manager written in Java. Both of them introduce a new feature in the software, and they both contain a (different) bug. The bugs have a functional nature (i.e., they affect the usage of the program from a user's perspective), therefore we ask you not to focus on code style and implementation details.

Note: you have to perform the review within the specific environment you have been assigned to. It is not allowed to download the code base locally, run the program, use an IDE or other code review environments.

Task 1

The pull request that you will be reviewing in the first task has the following description, as given by its author:

JabRef allows to rank each of the references in the library with a rank from 1 to 5 stars. This pull request introduces a new rank, rank 0, which corresponds to 0 stars.

Your goal is to find the functional bug in this pull request. There is a time limit of **10 minutes**. You can start the task clicking on the following link:

<https://github.com/joined/jabref/pull/3/files?diff=unified>

Task 2

The second pull request has the following description:

JabRef has a feature to export selected references from the database to a HTML document. The user can choose whether to export all the references in the database or only the selected ones. Currently there is no default file name for the exported HTML document, the user has to provide one. This change introduces the following feature: when a single reference is being exported, the default name of the exported HTML document is the citation key of the reference (e.g. Bacchelli2018.html). When multiple references are exported, the default filename is ExportedReferences.html.

Your goal is to find the functional bug in this pull request. **There is a time limit of 30 minutes.** You can start the task clicking on the following link:

<https://github.com/joined/jabref/pull/2/files?diff=unified>

A.4 Tool Evaluation: Treatment Group

Welcome to this survey on the evaluation of my master's research project.

My name is Lorenzo Gasparini and I am a MSc student in Computer Science at TU Delft. For my master's thesis, supervised by Prof. Alberto Bacchelli (ZEST, University of Zurich) and M. Eng. Tobias Baum (Leibniz Universität Hannover), we are researching the usage of visualisation techniques to improve cognitive supports in the code review process.

We developed a code review tool that enhances the classic GitHub pull request diff-view enabling an intuitive navigation of the context of the changed code. As understanding the context of the changes is often one of the first steps in code review, we hope to enable a more effective and quicker overall process.

The goal of this survey is to evaluate the effectiveness of the tool that we built in real-world usage scenarios. Your help is appreciated.

A.4.1 General questions

Same questions as for the control group, reference: A.3.1.

A.4.2 Tool introduction

Next, you are allowed to explore our code review tool and familiarise yourself with its features. You have a maximum of **5 minutes** to complete this part.

Click on the following link to open an example pull request in our tool:

```
http://changeviz.tk/?url=https://github.com/square/retrofit/pul  
1/2789
```

A.4.3 Code review

Finally, you are assigned two code review tasks. They consist in the review of two pull requests to the JabRef project, a bibliography manager written in Java. Both of them introduce a new feature in the software, and they both contain a (different) bug. The bugs have a functional nature (i.e., they affect the usage of the program from a user's perspective), therefore we ask you not to focus on code style and implementation details.

Note: you have to perform the review within the specific environment you have been assigned to. It is not allowed to download the code base locally, run the program, use an IDE or other code review environments.

Task 1

The pull request that you will be reviewing in the first task has the following description, as given by its author:

JabRef allows to rank each of the references in the library with a rank from 1 to 5 stars. This pull request introduces a new rank, rank 0, which corresponds to 0 stars.

Your goal is to find the functional bug in this pull request. There is a time limit of **10 minutes**. You can start the task clicking on the following link:

<http://changeviz.tk/?url=https://github.com/joined/jabref/pull/3>

Task 2

The second pull request has the following description:

JabRef has a feature to export selected references from the database to a HTML document. The user can choose whether to export all the references in the database or only the selected ones. Currently there is no default file name for the exported HTML document, the user has to provide one. This change introduces the following feature: when a single reference is being exported, the default name of the exported HTML document is the citation key of the reference (e.g. Bacchelli2018.html). When multiple references are exported, the default filename is ExportedReferences.html.

Your goal is to find the functional bug in this pull request. **There is a time limit of 30 minutes**. You can start the task clicking on the following link:

<http://changeviz.tk/?url=https://github.com/joined/jabref/pull/2>

A.5 Tool Evaluation Results

Candidate	Q ₁	Q ₂	Q ₃	Q ₄	Q ₅	T ₁ (min)	T ₂ (min)	Group
P ₁	PhD	1 year or less	3-5 years	Once a day or more often	Once a week	4.50	Not found	Treatment
P ₂	Programmer / Software Dev.	3-5 years	2 years	Once a day or more often	Once a day or more often	5.50	Not found	Treatment
P ₃	Programmer / Software Dev.	6-10 years	6-10 years	Once a day or more often	Once a day or more often	4.00	23.00	Treatment
P ₄	Programmer / Software Dev.	3-5 years	6-10 years	Once a day or more often	Once a day or more often	3.00	13.50	Treatment
P ₅	Student	No experience	1 year or less	Once a day or more often	Once a month	6.00	Not found	Control
P ₆	Student	No experience	2 years	Once a month	Once a year	6.50	Not found	Control
P ₇	Student	No experience	2 years	Once a week	Once a week	2.50	Not found	Control
P ₈	Student	No experience	3-5 years	Once a day or more often	Once a week	6.00	Not found	Control
P ₉	Student	2 years	3-5 years	Once a day or more often	Once a week	7.00	Not found	Control
P ₁₀	Student	1 year or less	6-10 years	Once a week	Once a year	6.50	Not found	Treatment