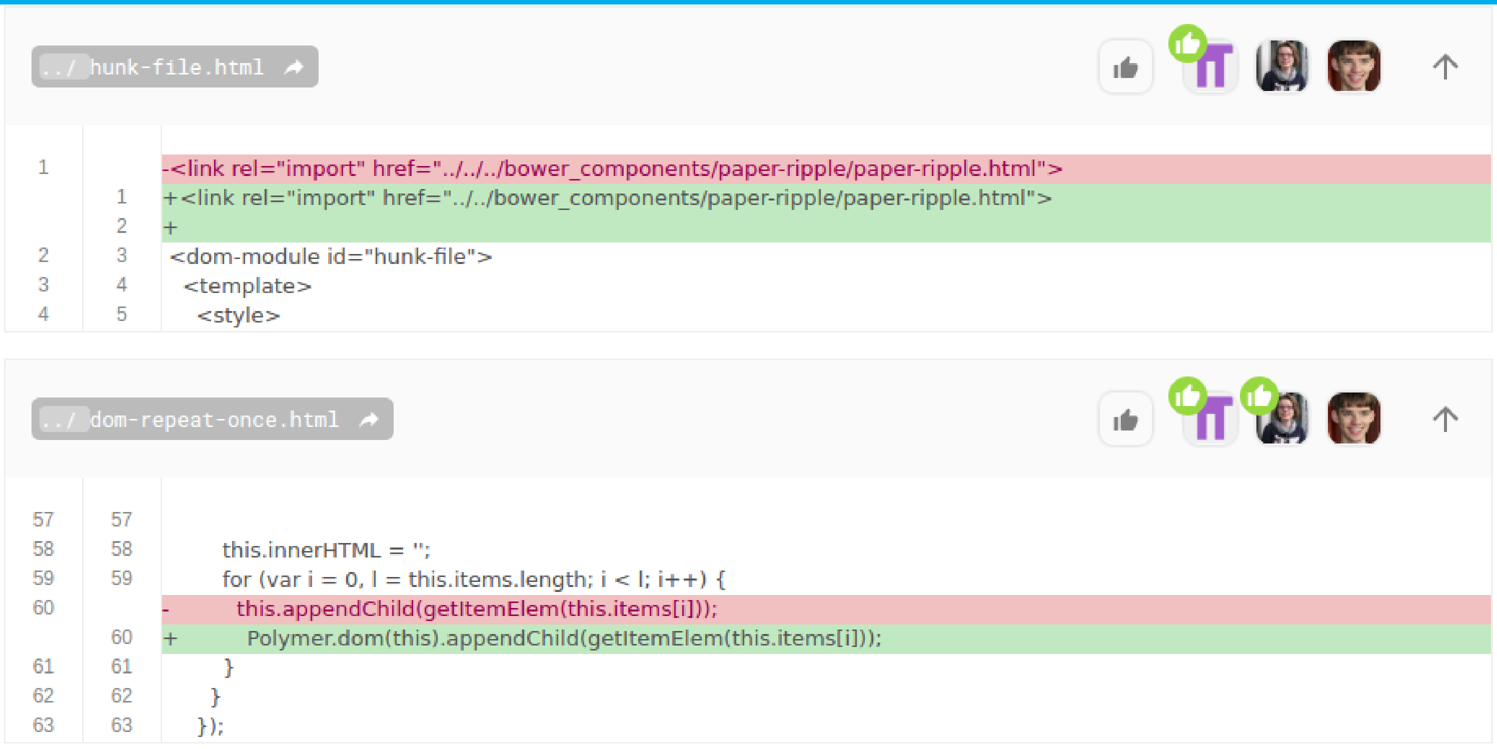
Enhanced GitHub code review

Eva Anker
Tim van der Lippe
Thomas Smith

Client Alberto Bacchelli
Coach Alessandro Bozzon



../ hunk-file.html

```
-<link rel="import" href="../../../bower_components/paper-ripple/paper-ripple.html">
+<link rel="import" href="../../bower_components/paper-ripple/paper-ripple.html">
+
 <dom-module id="hunk-file">
   <template>
     <style>
```

../ dom-repeat-once.html

```

       this.innerHTML = '';
       for (var i = 0, l = this.items.length; i < l; i++) {
-        this.appendChild(getItemElem(this.items[i]));
+        Polymer.dom(this).appendChild(getItemElem(this.items[i]));
       }
     }
   });
```

TUDelft

Enhanced GitHub code review

by

Eva Anker
Tim van der Lippe
Thomas Smith

Project duration:	April 18, 2016 – June 24, 2016	
Project committee:	Dr. A. Bacchelli,	TU Delft, company contact
	Dr. Ir. A. Bozzon,	TU Delft, coach

TU Delft

Contents

1 Introduction

Git repository hosting websites like GitHub and Bitbucket have invested significant effort into seamlessly integrating pull based development with their services. Since then, about half of all projects on GitHub that use some form of distributed collaboration use this pull based approach [Gousios et al., 2014]. Developers extend or modify existing source code and publish the set of changes in a pull request with the goal to integrate the changes into the project the pull request is created on. Tightly coupled to the submission and approval of pull requests is the process of reviewing proposed changes. A code review is an informal process by a maintainer of the project to check if the set of changes complies with the project coding conventions as well as judging the quality of the proposed implementation. Discussing different alternative solutions is common practice in code reviews. Empirical research showed that 75% of software project maintainers on GitHub conduct code reviews on every pull request submitted to the project, while 60% view code reviews as obligatory [Gousios et al., 2015b].

The main motivations for such reviews according to project maintainers is to find defects and ensure a baseline quality of proposed changes. This motivation is underscored by recent findings that correlate code reviews with decreased post-release defect counts [McIntosh et al., 2014]. While this is the most obvious reason to conduct a code review, the main benefit does not appear to be finding defects. Research conducted at Microsoft by Bacchelli and Bird shows that, while code reviews are somewhat successful in finding defects, most benefits stem from the communicative nature of such reviews. Among those are better knowledge transfer, increased team awareness and the proposal of alternative solutions to problems [Bacchelli and Bird, 2013].

From this we can see that the process of reviewing code involves many aspects of both technical and social nature. This makes building a system that smoothly integrates code reviews with pull based development a nontrivial task. Code review tools should neither be built around the singular goal of finding defects, nor should they simply be a tool that enables communication on top of pull requests. Tools should enable developers and maintainers to tap into each of the benefits of code reviews, while mitigating most of the challenges that come with these reviews.

The current state of commonly used review systems such as the one provided by GitHub are not sufficiently mitigating the challenges. In section 2 these challenges with code reviews are discussed as well as the shortcomings of the GitHub user interface. We derived several requirements from the challenges discovered which are listed in section 3. A motivation is included for several technology and tool choices. The requirements are translated into design choices, as described in section 4. The design was evaluated iteratively using the evaluation method Rapid Iterative Testing & Evaluation (RITE).

Before implementing the architecture of the system needs to be thought out. The high level overview of the system, including the data flow concerning the external APIs is elaborated on in section 5. After constructing the architecture of the system, the system is implemented. During the implementation some problems arose and SIG feedback concurring the code quality was received. This process is explained in section 6.

The product needs to be evaluated, by measuring evaluation metrics and employing a quantitative evaluation. Section 7 describes metrics and an accompanying experimental protocol for a quantitative evaluation. A conclusion is drawn in section 8 and after that a reflection on the project is included in section 9. Lastly, some recommendations will be given in section 10. These recommendations are aimed at developers who continue to work on this project.

2 Problem definition and analysis

This section provides a clear problem definition and lists all accompanying challenges.

2.1. Problem definition

Current pull request interfaces (e.g. GitHub and Bitbucket) provide a visual representation of the changes corresponding to the pull request. The visual representation incorporates code review by enabling users to post comments on the textual diff of the changes (inline comment) and on the pull request as a whole (generic comment). All comments are aggregated on the main pull request overview ordered primarily by time of posting. Comment threads can be created by adding a comment to the corresponding line of an inline comment.

The interface of the main pull request overview leads to problematic communication between the integrator (reviewer of the pull request to integrate the changes into the project) and the contributor. The overview not only shows comments, it also depicts label changes (to indicate the status of the pull request), change of assigned reviewer and commits which are added to the pull request. As a result of a problematic communication, pull requests are improperly reviewed. Improper and hastily reviewed pull requests lead to a lower software quality [McIntosh et al., 2014]. McIntosh et al. show that when a low percentage of the submitted code is reviewed, the corresponding components tend to have a high number of post-release defects.

We observe that the interface of the pull request overview as well as the textual diff results in low quality code reviews. Consequently low quality code reviews result in more post-release defects. An improved UI for the code review process is required to maintain high-quality software.

2.2. Problem analysis

We analyzed the current approach of code review on GitHub and associated literature on the modern code reviewing process. Challenges in this process as well as problems in the user interface facilitating code reviews were discovered.

Understanding the changes in a pull request Reviewers indicate that understanding the reasoning behind changes in a pull request is the number one challenge when processing a pull request [Bacchelli and Bird, 2013]. A lack of context for the changes is the most prominent problem when reviewing a pull request. Examples of lack of context are incomplete descriptions, the reviewer is unfamiliar with the files changed or a complete overview of the impact of the changes is missing/incomplete.

Abandoned pull requests remain open Pull requests that are insufficient in terms of technical quality or pull requests that do not fulfill mandatory requirements are left open [Gousios et al., 2015b]. As a result of lack of response of the original contributor, the pull request is abandoned and not closed. On the other hand, contributors indicate that low responsiveness of reviewers causes frustration and contributors are not willing to continue to work on the pull request [Gousios et al., 2015a]. When a reviewer is not willing to merge the pull request, instead of leaving the pull request open without a response, contributors prefer to have a clear reject.

Only one reviewer can be assigned at a time Large projects with multiple integrators review pull requests with two or more reviewers [Gousios et al., 2015b]. At the moment of writing, the interface allows only one person to be assigned to a specific pull request.

Availability of reviewer is a limiting factor In conjunction with the previous points, the availability of the reviewer is a clear indicator for the time to merge a pull request [Gousios et al., 2015b]. If a reviewer has to review multiple pull requests simultaneously or in a quick fashion, the reviewer requires relatively more time to get familiar with the files touched in the pull request and the corresponding context. The time required per pull request increases which results in fewer processed pull requests per time unit. This problem is especially prominent in teams with few integrators.

Multiple features in one pull request Implementing a large feature affecting multiple areas makes reviewing hard [Gousios et al., 2015b]. A direct problem is feature isolation: the pull requests solves multiple issues at once where the review partially rejects the pull request. This results in an unmerged pull request, whereas a portion of the pull request was accepted and could have been integrated into the project. There is no clear communication method on what parts are accepted and what parts are rejected.

Pull request overview The pull request overview currently shows all updates of the pull request in one view. All comments of the code review are linearly displayed ordered by posting time. Both the reviewer and the contributor quickly lose track of the current status of the pull request and comments that should be or have been fixed. This is especially prominent if multiple people comment on different portions of the pull request.

Lack of team awareness Large pull requests contain a lot of changes which the team should be aware of. If a large pull request is inadequately reviewed, a significant portion of the changes remains unchecked. Therefore code is integrated into the product which the team has marginal awareness of. Team awareness is one of the reasons code review is applied in project teams [Bacchelli and Bird, 2013].

3 Requirements

The goal of this project is to build a tool, that displays the code changes in a more dynamic and intuitive way. To make the tool, requirements needed to be defined together with the client. These requirements are detailed next. The requirements are separated into two types: functional and non-functional. Additionally the functional requirements are defined in the four categories of the MoSCoW model: Must-haves, Should-haves, Could-haves and Won't-haves. Lastly, we provide a definition of done to list what a specific requirement should fulfill before it is considered done.

3.1. Functional requirements

Must-haves

R_M1 The system shall be compatible with the GitHub REST-API[1]

R_M2 The user shall log in with GitHub OAuth[2]

R_M3 The user shall be able to view all their public and private repositories

R_M4 The user shall be able to view all pull requests in the currently selected repository

R_M5 The user shall, as a contributor, be able to create a pull request

R_M6 The user shall, as a contributor, be able to create groups for his/her pull request

R_M7 The user shall, as a contributor, be able to order the groups and code hunks in a pull request

R_M8 The user shall be able to organize all groups of the currently selected pull request in a 2D space

R_M9 The user shall be able to view and place comments on a pull request

R_M10 The user shall be able to view and place comments on a group

Should-haves

R_S11 The system shall be compatible with the Bitbucket REST-API[3]

R_S12 The user shall log in with Bitbucket OAuth[4]

R_S13 The user shall, as contributor or reviewer, be able to set the status of a pull request

R_S14 The user shall, as a reviewer, be able to mark a hunk as reviewed

[1] https://developer.github.com/v3/
[2] https://developer.github.com/v3/oauth/
[3] https://developer.atlassian.com/static/rest/bitbucket-server/4.5.2/bitbucket-rest.html
[4] https://confluence.atlassian.com/bitbucket/oauth-on-bitbucket-cloud-238027431.html

R_S15 The system shall be able to provide a list of unreviewed hunks

R_S16 The user shall, as a contributor, be able to create links between groups in their pull request

R_S17 The user shall be able to view a list of pull requests that require their attention

R_S18 The user shall, as a contributor, be able to view a busyness indicator of the reviewer

R_S19 The user shall, as a contributor, be able to assign reviewers to their pull request

R_S20 The user shall, as a contributor, be able to view a code ownership metric for each reviewer

R_S21 The system shall display the amount of reviewers that have reviewed the pull request

R_S22 The user shall be able to place replies to existing comments on a pull request or group which are ordered hierarchically

Could-haves

R_C23 The system shall be able to propose reviewers automatically based on code ownership and reviewer availability when a user creates a pull request

R_C24 The system shall be able to automatically propose an organization of hunks and groups when a user creates a pull request

R_C25 The user shall be able to search for public repositories

R_C26 The system shall incorporate continuous integration feedback

R_C27 The system shall provide the pull request template[5] when the user, as a contributor, creates a pull request

R_C28 The system shall provide quick access to the issues and pull requests on corresponding external sites

R_C29 The user shall be able to quickly view the whole file, when clicking on a hunk

Won't-haves

R_W30 The system shall automatically manage pull requests without human intervention

R_W31 The user shall be able to modify code using this system

R_W32 The system shall perform static analysis of code

R_W33 The user shall be able to create and store repositories on the system

R_W34 The user shall be able to create an issue on a repository

3.2. Non-functional requirements

R_{NF}35 The code shall be versioned with git[6]

R_{NF}36 The code will be developed in a shared repository using the pull request model. A feature will be developed in a branch and a pull request will be made when the feature is completed

R_{NF}37 For the project, the Scrum[7] methodology shall be applied

R_{NF}38 The backend will be implemented in Java[8]

R_{NF}39 The frontend will be implemented using the Polymer library[9]

[5]https://help.github.com/articles/creating-a-pull-request-template-for-your-repository/
[6]https://git-scm.com/
[7]http://scrummethodology.com/
[8]http://openjdk.java.net/
[9]https://www.polymer-project.org/1.0/

$R_{NF}40$ Frontend and backend shall communicate via a REST-API

$R_{NF}41$ Continuous integration is used to verify all tests pass on the pull request

$R_{NF}42$ A predefined code formatting will be maintained using Checkstyle[10] to prevent auto-formatting issues and ensure consistent coding style

3.3. Definition of done

A feature of the product is considered done when all of the below points hold for that feature.

- Code changes were reviewed by at least one other team member
- Code changes were incorporated in the master branch
- All tests pass on continuous integration
- 80% test coverage is measured on the new feature
- At least two team members have manually tested the feature

3.4. Motivation choice of technologies

In the non-functional requirements we listed several technologies we will use in our project. This section will briefly reflect on the motivation for these choices.

Requirement 38 states the backend will be implemented in Java. Java is the programming language all team members are most familiar with. We have written similar applications in Java and have extensive knowledge on existing used frameworks, libraries and coding conventions.

Similar to our experience with Java, we will implement the frontend using the Polymer library as stated in requirement 39. There is a significant amount of recent and relevant experience with sites developed using the Polymer library including contributions to the library itself. The Polymer library enables developers to write small and simple reusable components, also known as custom elements. These components can be composed to build a hierarchical structure of an application.

The method for the frontend to communicate with the backend is a REST-API (requirement 40). Last year during the context project all team members implemented such an API and in more recent projects applied this methodology.

3.5. Motivation choice of tools

The project process is assisted with the usage of several tools to improve the productivity and enhance the communication between the team members. Following is a list of tools we will use:

GitHub The project source code is governed by version control (requirement 35) and hosted on GitHub[11]. As our project aims to improve the GitHub UX, we will obtain hands-on experience by using our product directly on our own development and review process.

Travis The continuous integration (requirement 41) is handled by Travis[12]. Our experience is that the service seamlessly integrates with GitHub by running the tests in the project whenever a branch or pull request is created or updated. We will therefore have near-instant feedback on the correctness of the implemented functionality.

Waffle.io Our scrum process (requirement 37) is assisted with the usage of Waffle.io[13][14]. Waffle.io integrates the scrum process into GitHub issues by enabling the team to create a backlog of ordered issues. This way issues can be ordered by priority to highlight most-pressing issues from less important issues. Additionally, points can be added to issues to indicate the amount of time required to develop the designated feature or fix the designated

[10] http://checkstyle.sourceforge.net/
[11] https://github.com/preview-code
[12] https://travis-ci.org
[13] https://waffle.io/preview-code/frontend
[14] https://waffle.io/preview-code/backend

bug. We can then get a quick overview of the number of issues we can fix in a given week and provide a meaningful overview for the client on what to expect for the next demo.

Slack We have direct communication with our client using Slack[15]. This medium is primarily used to share research ideas and share relevant research papers for our project.

3.6. Licensing

Before developing our product, we have looked for a suitable license for our project. Since the project is intended to improve the user experience, we as users and developers want to use the technology to enhance our own experience as well. Therefore we needed to choose a suitable license that allows us to develop the product, while allowing other parties to improve the product and offering it as a service which we can use.

3.6.1. Potential licenses

GPLv3 The GPLv3-license has relatively strict conditions. Everyone may copy, distribute and modify the software as long as the source code is public and is published under the GPL-license as well. Additionally, significant changes in the software must be stated. The consequence is that if the source code is licensed by GPL then all products using the software should be licensed by GPL as well.

LGPLv3 The LGPLv3-license is equal to the GPLv3-license, but without the notice that the work that links to the software does not fall under the must be LGPL restriction.

BSD 3-clause The BSD-license permits parties to distribute the source code of the product, with or without modifications, with the only requirement that the source code should include the copyright notice of the original authors. The consequence is that the product may be improved upon and distributed as long as the original authors are cited in the source code. Accepted distribution includes developing proprietary software and publish it as a service.

MIT The MIT-license is very similar to the BSD-license, with the exception that the original copyright notice is not required to be added to the redistribution of the original source code. Authors of modifications to the software are therefore free to choose their respective license and profit model.

Apache 2.0 The Apache-license is also very similar to the BSD-license, with the added patent notice of the original authors. Therefore this requires us to patent the product and distribute the patent in the license.

3.6.2. Conclusion

After evaluating all consequences of each license, we chose the BSD 3-clause license. Since we want to receive credit for the work described in this paper, a permissive license such as the MIT-license does not satisfy the requirements. We have not patented our idea however which makes us unable to use the Apache 2.0 license. This left us with the choice between the GPL-licenses and the BSD-license.

Our second requirement previously stated was the permission for other parties to improve the software. A GPL-license would greatly disincentivize the development of the product, as the license is not suitable for companies who want to make profit with their improved software product. Therefore the last option is the BSD-license, which does allow companies to improve the product, while leaving the original credit for us as the authors.

[15]https://slack.com/

4 Product design and RITE

4.1. Design

In this section the design of the product, called PReview-Code is elaborated on. The design choices are explained, with references to the challenges discovered in section 2.

4.1.1. Groups

To split up the line changes of a large pull request, the user can create groups. These groups are composed out of hunks. A hunk is an isolated block of related lines. A similar feature has already been implemented in an IDE [Bragdon et al., 2010a] and the effectiveness of this feature is already proven [Bragdon et al., 2010b].

Ordering of groups

Groups can be ordered to emphasize the importance of the hunks inside relatively to accompanying hunks in the pull request. Inside a group the hunks can be ordered as well, to make a distinction between importance inside a group. Following the logical order of the contributor improves the understanding of the integrator when reviewing the pull request.

Connection among groups

Additionally, groups can be connected to other groups to emphasize that the groups are related to each other. The contributor can also add code snippets as a hunk and connect it with the existing group. This is for example useful to show usage of the API developed or to show in which situation a bug was fixed. The whole file where the hunk is located is also easily accessible. By adding connections and context, the challenge of understanding the changes in the pull request is partially solved.

Review checklist

In conjunction with the ordering and connections of groups, a checklist can be added to the pull request. The checklist denotes which hunks of the pull request have been reviewed and accepted. This makes it easier to distinguish the multiple features in the pull request. The challenge of communicating about accepted and rejected portions of the pull request is solved.

4.1.2. Comment hierarchy

Every pull request has a comment hierarchy for every comment that is placed on a group or the pull request itself. The comment hierarchy structure denotes the relation between a pair of comments. Since the comment hierarchy is linked to a group, the pull request overview is less cluttered. Moreover, discussions are not linearly structured, which makes reading the comment thread easier. The corresponding context to a reply is directly visible in the hierarchy above the specified comment.

4.1.3. Recommend reviewer

On every pull request multiple reviewers can be assigned. The appointment of corresponding reviewers can be automated to ensure that the correct reviewers are designated to pull request. This selection ensures the reviewer is reviewing a pull request he/she has a lot of knowledge about, which as a result increases the review quality [Bacchelli and Bird, 2013]. This partially solves the problem of understanding the pull request, because the person who is reviewing knows a lot about the subject. Lastly, it partially solves the problem of the limited availability of reviewers; the reviewer can spend their time in such a way they review pull request they do not have to invest a lot of time in getting to know the files touched.

4.1.4. Status

Every pull request can have a status that shows the current state of the pull request. This can be set by both the contributor as well as the reviewer to enable bi-directional communication. Examples of statuses are ”No reviewer assigned” (set by contributor) or ”Awaiting contributor feedback” (set by reviewer). This solves part of the awareness problem because it is easier to find which pull request needs attention and allows reviewers to quickly communicate a status update to contributors. From the contributors perspective, this relieves the problem of having to wait a long time without knowing what the status of a pull request is.

4.1.5. Workload indicator

Every reviewer will have a indicator visible for contributors which displays the current workload of a maintainer. This enables the contributors (and also the automatic reviewer selection algorithm) to take busyness of integrators into account. The goal of this feature is to distribute reviewing load such that not one maintainer has a significantly higher reviewing load than other maintainers, relieving the common problem of maintainers experiencing stress from high workloads.

4.2. Rapid Iterative Testing and Evaluation (RITE)

To come up with a good design for our final product, the RITE method is used to iteratively evaluate prototypes of PReview-Code. RITE is a method to identify problems in an interface, and evaluate the efficacy of the applied fixes to the problem [Hanington and Martin, 2012]. The evaluation is performed by letting participants comment on the design and possibly list design problems. When a certain number of consecutive participants do not list any design problem, the final design is obtained.

In other words, an initial design is proposed and tested by the first participant. Potential problems are identified and solutions to the problems are developed. The new design including all developed solutions is shown to the next participant. This process will continue until the determined amount of participants do not list any design problem.

We chose to employ RITE, because ”The goals of the RITE method are to identify and fix as many issues as possible and to verify the effectiveness of these fixes in the shortest possible time” [Medlock et al., 2002].

During the RITE process we decided to keep evaluating new versions of PReview-Code with RITE. All developed features will be evaluated by new RITE participants. As a result not only the first version is evaluated, but also all new versions of PReview-Code ensuring all features are usable.

4.2.1. Prototyping tool

For the RITE evaluation we had to build a prototype. We needed a design tool that allows us to easily make changes based on the feedback we received, as well as a usable prototype to mimic the real interaction as close as possible.

Possible tools

Axure Protoypes built with Axure can be as realistic as possible, with functions including hover over, clicking and routing. However, it is impossible to create a realistic design for our product with Axure alone because that would require images developed in an external program.

Illustrator Illustrator is a tool in which static graphic designs can be made. To mimic the real interaction as close as possible with real-time feedback, Illustrator must be used in conjunction with a tool like Axure. Currently, only one team member has the skills to work with Illustrator.

Polymer designer Polymer designer is a drag and drop tool to make Polymer code and prototype designs. This tool only works with Polymer version 0.5 and therefore misses a lot of currently used components that would be used in our real application. The transition from the old to the new code would take a considerable amount of time and effort.

Polymer The prototype can also be made in Polymer itself. This requires that the designer creates the prototype with code, which takes more time upfront, but saves time when converting the prototype into a real application. An advantage of using Polymer is that the prototype will be identical to the real UI. This approach makes it easier to create an interactive prototype while reducing the amount of time needed to transform to a working application.

Conclusion

After evaluation the pros and cons of the tools, we chose Polymer. Since making changes in the prototype needs to be easy and quick, it is not feasible to depend on one team member, therefore Illustrator is not an option. The design needs to be as real as possible to get the most out of RITE, Polymer designer and Axure do not fulfill this requirement. Polymer does fulfill all requirements, and the extra start-up costs outweigh the benefits of transitioning to the real application more easily.

4.2.2. Approach

In this section we will describe our approach of applying the RITE evaluation to our product.

Participants

The audience of our application is reviewers, therefore we selected participants that review code on a daily/weekly basis. They all performed their reviews with an interface similar to the GitHub interface.

Measurements

The data source of the evaluation was the communication with the participant and the interaction with PReview-Code. This data source can be more specified as in, the answers to the questions, the (non)verbal communication, the way the participant performed the task, the completion of tasks and lastly the answers to the question.

The errors from the data source can be divided into multiple categories.

Recommendation The participant can perform a task, but states that the process could be improved.

Confusion The participant is confused about the working of a certain part of the system. For example clicking the wrong button. After a while the participant is able to continue their task.

Non-understanding The participant is unable to correctly answer a question.

Non task completion The participant is unable to complete a task.

Procedures

To evaluate the product, a story was made to simulate the real workflow of the participant. This story was designed in such a way that several tasks where included to be performed by the participant. Tasks were accompanied with questions that should assess whether the participant understood what he or she did. Moreover, we asked participants to think aloud to follow their thought process and discover at what moment an incorrect decision is made.

Incorrect decisions were categorized into 3 categories:

1. The issue appear to be a blocking issue for new users. These issues must be fixed before a new participant uses the product.
2. Issues which need to be improved, but are not of a blocking nature. Such issues are inconvenient and should be fixed, but participants can use the product nonetheless.
3. Lastly all issues that a participant discovered, but have to be further investigated if more participants label this as an issue. Examples of such issues are mostly styling related, e.g. the style of a piece of text or the placement of an icon.

The following tasks needed to be performed by the participant:

- View a specific pull request
- Reorganize groups and hunks in this pull request
- Find a specific comment in a pull request
- Create a pull request

The following questions accompanied the above tasks:

- How many pull requests are open at project X?
- Who was the creator of the pull request?
- Who is the first commenter on the pull request?
- How many comments are posted on this group?
- How many reviewers have accepted the changes?
- Why is this pull request still open?

Results

During the RITE evaluation multiple errors where discovered. In table 4.1 the amount of errors are shown, with the classification and the amount of fixed errors. Duplicate errors are filtered, because some issues could not be solved before a new participant was evaluating the product.

Table 4.1: Classification of errors

	Recommendation	**Confusion**	**Non-understanding**	**Non task completion**
Errors	25	5	0	3
Fixed	22	5	0	3

The three tasks that could not be completed were related to the group and hunk list. The first participant did not know what hunks where, and therefore could not complete this task. After a better definition of a hunk, the next participant was unable to reorganize the hunks, because the dragging and dropping hunks was not intuitive. Lastly a participant had difficulties with a sub task, namely the collapsing of hunks. After a lot of changes to the interface every participant was able to complete the tasks.

One of the tasks that caused confusion was the creation of a pull request. Whilst the pull requests of a project where shown in a list on the left side, the button to create a pull request was positioned on the right side of the screen. When the participant finally found the button, he/she admitted they did not expect the button to be placed on the right side of the screen. Their eyes were fixed on the list itself. After putting a button in the pull request list, every participant could perform this task without confusion, but no participant has ever pushed the old button. Another error that was classified confusion was the hiding of the project and pull request lists. These lists where both shown on the page and could be hidden with an arrow, but the purpose of this arrow was not that clear for the participants. We have replaced the arrow with a tab-structure which was approved by all consecutive participants.

Another frequent comment was to add more tooltips. It turned out that not every icon was as clear as we thought and therefore all icons now have a tooltip. Another recommendation was to have a better integration with other services. For example, the Continuous Integration (CI) status was shown, but it was not clickable. We changed the behavior to be able to click on the CI status to go to the log of the corresponding build in the tool. Also the integration with GitHub was improved, if an issue is referenced this will be rendered as a link to the corresponding issue.

From all errors 90% was fixed, only a few recommendations where not fixed. These recommendations were not really errors, but more feature requests. These were features we would like to implement, but unfortunately there was no time to incorporate these features. An example of such a feature, is to show if a branch could be merged into the head branch without merge conflicts upon the creation of a pull request.

5 Architecture of the system

This chapter will elaborate on the high level overview of the architecture of the system. In particular, the information flow concerning the external APIs is explained and how the user interacts with the system.

5.1. Overview

In figure 5.1 the dataflow between the components of the system and external APIs is shown. In the coming sections, every component of the diagram will be elaborated on. The different components are the client which is the website. The backend which performs the calls to the database and GitHub. GitHub is the service we use to get the data from the repository. Lastly Firebase functions as the database for data that is not stored on GitHub.

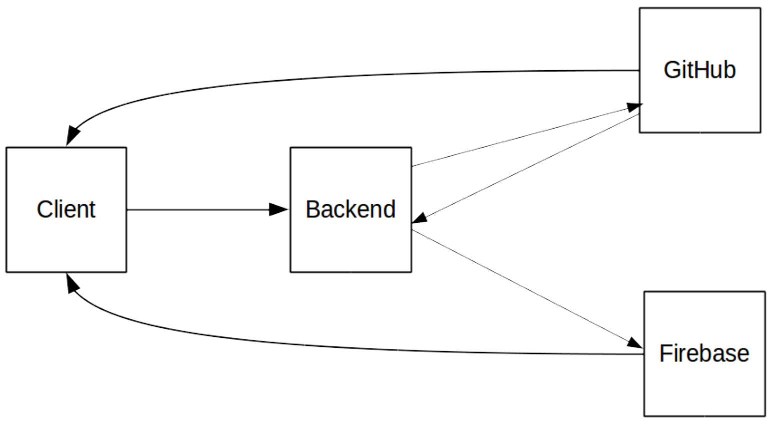


Figure 5.1: The dataflow between the system and external APIs

5.1.1. Clientside

All gathering of user input and visualization is performed on the clientside. The clientside is the website written in Javascript using the Polymer library. Communicating with external APIs is based on AJAX-requests.

Retrieving data from GitHub All read operations on the GitHub API are processed by the element `github-authenticated-ajax`. The sole response of this element is to perform GET HTTP requests on the GitHub API authenticated with an oAuth token. This oAuth token is retrieved from the Firebase authentication as described in the section below. In conjunction of authentication, the element performs ETag management to make efficient use of the rate limit employed by GitHub. Section 6.3 describes this process in more detail.

Retrieving data from Firebase All data that we store in our Firebase database is automatically synced to all connected clients. The clients are connected via the `firebase-document` element built by the Polymer team [1]. This element opens a websocket connection to the Firebase database. Whenever data is modified in the Firebase database, the websocket connection is notified of the change. Consequently the newly retrieved data is exposed to the Polymer data-binding system by the `firebase-document` element. As a result, when a user modifies for example the ordering of the pull request changes, all other users connected to this pull request adopt the new ordering in real-time.

Posting data to the Java backend All post operations on the backend are processed by the `backend-ajax` element. This element performs POST HTTP requests on the backend with the oath token of the user. This oath token is retrieved from firebase, as described in the section below.

5.1.2. Backend

In the backend calls from the client are processed. These calls are redirected to the right API, GitHub and/or Firebase. In the backend is also verified if the user is allowed to post this data. The backend is writen in Java, using a REST-API.

Retrieving data from the client From the client POST requests are made to the Java backend. All this data is supposed to go to GitHub or Firebase, according to the request.

Posting data to GitHub From the backend POST requests are made to the GitHub API, this will update the repository data on GitHub.

Post response from GitHub After a POST request to the GitHub API, a response gets back. In this response is stated if the request is accepted.

Posting data to Firebase If the data that the backend needs to process is send to Firebase and GitHub. The data is only send to Firebase when the GitHub API sends a good response. With the POST request from the backend to the Firebase API data is written onto the Firebase database.

5.1.3. GitHub API

On GitHub all data is stored that is normally stored there, to make sure that a pull request is still reviewable at GitHub. From the backend and client calls are made to the GitHub API, to retrieve and write data.

Posting data to the client From the Client GET-requests are made, to retrieve data from GitHub.

Posting data to the Java backend From the Java backend GET-request are made in which is verified if the user is authorized to do a certain request.

Retrieving data from the Java backend From the Java backend POST-requests are made to the GitHub API. The calls are made to write data on GitHub.

[1] https://github.com/firebase/polymerfire

5.1.4. Firebase

Firebase serves as a database for the data that is not stored in on GitHub. The data is posted via the backend and retrieved via the frontend. This data is saved in JSON[2].

Initially we wanted to build the database with Hibernate[3] on the Java backend. However, this implementation would have several major challenges accompanied with it. First of all the setup required us to maintain the database, services and corresponding REST end-points. Secondly we wanted our application to be updated in real-time, which would require us to program some sort of live update system such as websockets. Given the time-constraints and the technical challenges, we looked into ready-to-use alternatives.

Our solution was instead to integrate Firebase into our application. Firebase is a real-time database with data-storage and live updates for all subscribed clients. If one user modifies the data in the database, all subscribed clients are automatically updated to the new values and can immediately show the new changes.

Since Firebase is hosted as a service by Google, it took us very little time to set up. After we had made an account, we could immediately integrate the database with our application front-end.

Retrieving data from the Java backend From the Java backend POST-requests are made to the Firebase API. The calls are made to write data on Firebase.

Posting data to the client From the client a websocket connection is opened to the Firebase API to get data that is stored in the Firebase database. Whenever the data changes on Firebase the data will automatically be send to all connected websockets.

From the client a GET-request is made to request the oAuth token of the current user. When Firebase gets this request, it will make a call to GitHub to fetch the oAuth token, this oAuth token is then returned to the client via Firebase.

5.1.5. Interaction with the system

To show the interaction between the user and the system two sequence diagrams are created.

Creation of a pull request

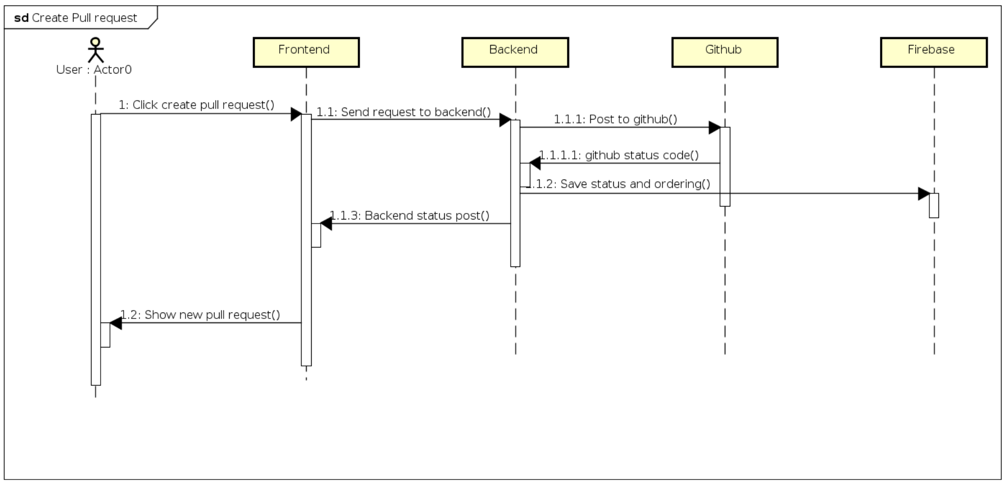

sd Create Pull request
User : Actor0
Frontend
Backend
Github
Firebase
1: Click create pull request()
1.1: Send request to backend()
1.1.1: Post to github()
1.1.1.1: github status code()
1.1.2: Save status and ordering()
1.1.3: Backend status post()
1.2: Show new pull request()

Figure 5.2: A sequence diagram over the creation of a pull request

In figure 5.2 the sequence diagram for the creation of a pull request can be seen. When a user clicks on the button 'create pull request' in the frontend the information about that newly made pull request is send to the backend. The backend will post that data to GitHub, and if it succeeds the data that can not be stored on GitHub, ordering of hunks and status, will be post to Firebase. The backend will tell the frontend it succeeded with posting the data and will give the data of the

[2]http://www.json.org/
[3]http://hibernate.org/

pull request. This data is then displayed in the frontend, so that the user can see the newly made pull request.

Adding an assignee

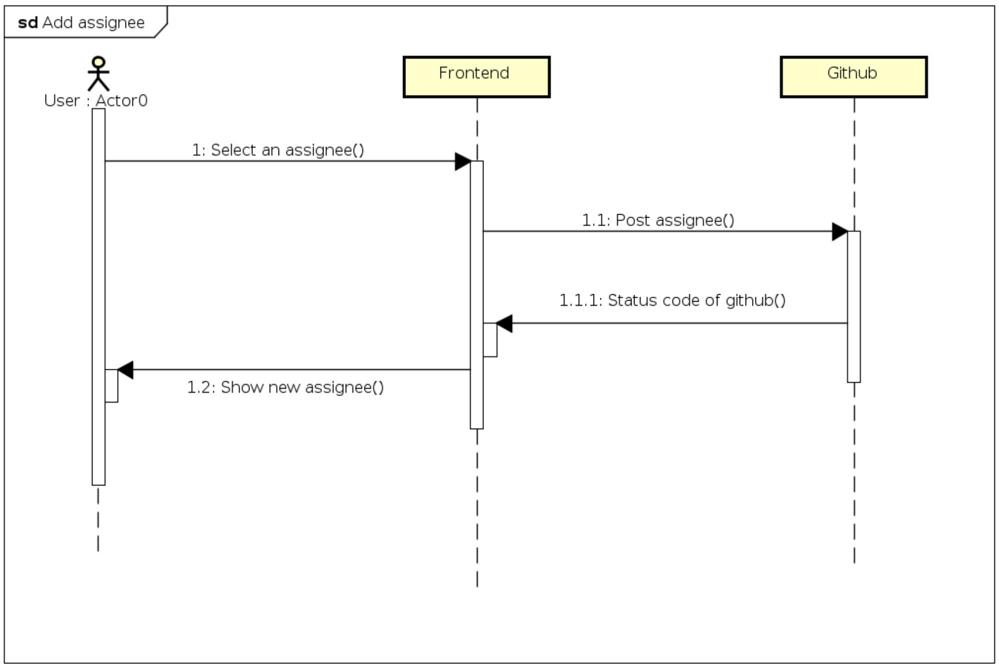


Figure 5.3: A sequence diagram over the adding of assignees

In figure 5.3 the sequence diagram for the adding of a assignee to a pull request can be seen. When a user selects an assignee, the data is sent to Github. If Github accepts this assignee, the assignee is shown in the frontend. Only Github is needed, because all data can be stored on Github. This is possibly because during the project, Github added the ability to add multiple assignees.

5.1.6. Datamodel

The data on firebase is stored in JSON format. The model can be seen in figure 5.4. For example, every comment is only shown in one group. So, if you get the array of comments, you get the id of the comment and the id of the group together. This way the data is easily accessible and ready to use.

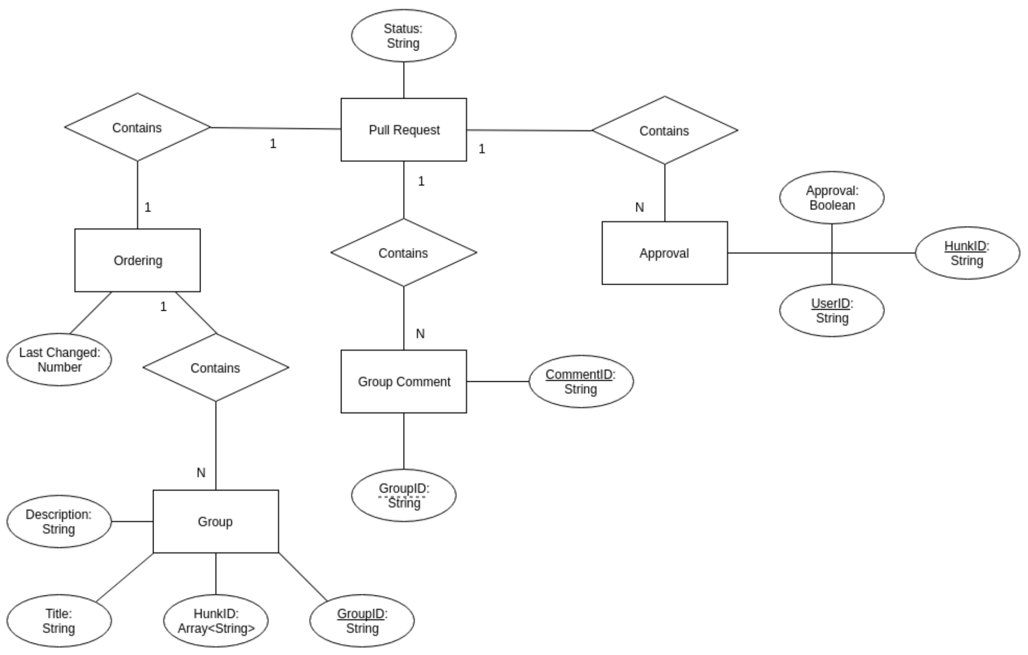


Figure 5.4: The datamodel of the database

6 Implementation of the system

In this section we will cover the most important implementation problems, solutions and decisions encountered while building the system.

6.1. Hunk tracking over time

The system enables contributors to organize hunks inside a pull-request. This ordering of changes is specified when the pull request is created. However, in the context of modern code reviews a pull request is not a static entity. It evolves over time with reviewer feedback and/or contributor improvements to the code. Revising code structure after a pull request was opened is commonplace in the open source world. From this, an interesting question about the ordering of changes arises; what should happen with the contributor-made ordering when a pull-request receives commits after the ordering was constructed? When a user commits after an ordering was made, this ordering may or may not become invalid. Changes that were ordered by the contributor may be reverted, moved, edited or replaced entirely by a new commit. One could simply remove the old ordering from the system when a new commit is detected, but this would place a significant burden on the contributor who would then have to re-organize all their changes. Doing this would discourage the user either to order their changes, or to revise their changes after creating the ordering. Instead of simply discarding an outdated ordering, we try to match the previously ordered changes with the changes made afterwards. By tracking changes over time we are able to automatically update the user-defined ordering most of the time. This process cannot be 100% accurate. Take for example the following situation where a contributor creates a pull request with the following change:

```
src/create-pull-request.html
@@ -29,2 +29,6 @@
 },
+_fetchBranchesAndCollaborators: function() {
+  this.$.ajaxBranches.generateRequest();
+  this.$.ajaxCollaborators.generateRequest();
+},
 attached: function() {
```

The contributor has added a function `_fetchBranchesAndCollaborators` to the file `create-pull-request.html`. Imagine the contributor has grouped this change with several other changes, and then commits the following three changes:

```
src/create-pull-request.html
@@ -29,6 +29,5 @@
 },
-_fetchBranchesAndCollaborators: function() {
-  this.$.ajaxBranches.generateRequest();
-  this.$.ajaxCollaborators.generateRequest();
-},
+_fetchPullRequests: function() {
+  this.$.ajaxPulls.generateRequest();
+},
 attached: function() {
```

```
src/pull-info-view.html
@@ -122,2 +122,5 @@
 },
+_getBranches: function() {
+  this.$.ajaxBranches.generateRequest();
+},
 _getPulls: function() {
```

```
src/project-info-view.html
@@ -100,2 +100,5 @@
 },
+_getCollaborators: function() {
+  this.$.ajaxCollaborators.generateRequest();
+},
 computeComments: function() {
```

A human might be able to interpret the intent of this change. Judging from these three diffs, the `_fetchBranchesAndCollaborators` function was split into two functions and moved to different files. However, there is no way, even for a human developer, to be 100% certain by just looking at the diffs that this was what the contributor intended. Maybe the removal of `_fetchBranchesAndCollaborators` and addition of the two other functions are completely unrelated. Another problem is introduced in the first hunk of this diff. The `_fetchBranchesAndCollaborators` function was removed, and a different function was added in it's place. Now should this hunk be placed in the same group as it was before? And where do we group the other two hunks? Only the contributor can know in this case how the ordering should be updated.

6.1.1. Solution

Automatically classifying new changes into existing groups has proven to be a difficult problem. Existing research in tracking relied on complicated and computational heavy Abstract Syntax Tree differencing algorithms [Falleri et al., 2014]. Such a solution is not usable in PReview-Code as (near-)instant responses are required when the user requests a page. An interim-solution must be found which is not capable of detecting all changes 100% of the time, but provides a decent success rate to assist the user most of the time.

A possible solution would be to apply a heuristic on the line numbers of the textual diff. Every diff hunk is accompanied by a header denoting the starting line-number and number of changed lines. For example `@@ -29,6 +29,5 @@` denotes that starting from line 29, 4 lines were removed (1 line context before and after) and starting from line 29, 3 lines were added (with the lines context remaining untouched). A new commit could change the lines in the same region and have the header `@@ -29,6 +29,4 @@`. We can now deduce that of the original diff, the same lines are removed but there was 1 line fewer added. It is very likely that these two headers point to the same original diff hunk.

Concluding, a heuristic based on changes in line numbers seems feasible. Future work should iteratively test the heuristic and fine-tune the boundaries for a large corpus of textual changes.

6.2. Hunk visualization

To be able to visualize hunks in a 2D space (requirement 8), we investigated several libraries and external tools. The requirements of the library/tool are that: 1. it should visualize hunks in a 2D space and show connections between hunks 2. the user must be able to edit the hunks and its connections.

We have analyzed several existing Javascript libraries that have built-in solutions that satisfy both requirements as well as the native capabilities of HTML5.

D3.js D3.js is a powerful high-level library to visualize data in the DOM [1]. It is primarily focused on providing the best representation for a given data structure.

Processing.js Processing.js is another library to visualize data, however it is much more low-level and uses the native canvas element to draw the shapes [2].

Native canvas Instead of relying on a library which is backed by the native canvas element, you can also directly draw to the canvas element [3].

Conclusion

We have analyzed all alternatives and tried to implement a basic use-case with each alternative. While D3.js was very promising, due to a lack of examples and a very high learning-curve as well as a complicated syntax, we were unable to built a solution that was editable in an easy way. Then we tried out Processing.js and compared the usability with using the native canvas element directly. The performance loss of including the whole library did not weigh into the usability increase over the native canvas element. Since our use-case was fairly simple (we do not require advance data visualization), the native canvas element provided enough tools to do the job.

6.3. GitHub ETag management

GitHub employs a rate limit mechanism to prevent abuse of their exposed API. Whenever a request is performed to the API, 1 point is deducted from the users credit. If this credit reaches zero, consecutive requests are denied by the API. By default, authenticated users have 5000 credits per hour. However, while developing the application with full integration with the API, we encountered several times that we hit the limit.

To mitigate this problem, GitHub recommends implementing ETag management [4]. For every response of the API, an additional ETag header is sent with a generated hash. This hash is based on the content of the response. Whenever the content of the response changes, the hash is changed too. If an API request sent to GitHub contains the `If-None-Match` header, GitHub will compare the provided hash with the current hash. If a match was detected, the content of that specific end-point has not been changed and GitHub responds with a `304 Not Modified` status. Responses with a 304 statuscode do not deduct on the users rate limit. The advantage is therefore a severely reduced risk of hitting the rate limit as well as smaller responses which result in faster communication between the client and the API.

Since responses with a 304 statuscode do not contain the actual response, the client has to maintain a cache of the previous response. This cache is implemented in two places:

`github-authenticated-ajax` As explained in 5.1.1 all data retrieval of the GitHub API is performed with this element. The ETag management is implemented in this element as well, in an in-memory object cache. When the AJAX request is issued, the element checks an object cache for the corresponding end-point URL.

`service-worker` A `service-worker` is a relatively new technology that allows developers to implement custom network request processing. The main benefit of using a `service-worker` is in-between session caching. The cache of `github-authenticated-ajax` is

[1] https://d3js.org/
[2] http://processingjs.org/
[3] https://developer.mozilla.org/nl/docs/Web/API/Canvas_API/Tutorial
[4] https://developer.github.com/v3/#conditional-requests

a Javascript object and therefore limited to the current user session. The `service-worker` cache is persistent between sessions and therefore using a `service-worker` results in a lower consumption rate of the GitHub credits. The cache works just the same as with a regular `github-authenticated-ajax` and is there a progressive enhancement.

6.4. First SIG feedback

On Friday 3rd of June we received the feedback of the code submitted on Friday 27th of May. The e-mail (in Dutch) can be found in Appendix D.

The analysis of SIG focuses on the project structure and structure of single HTML files. They indicate that the average unit size of a file is high as a result of the Polymer coding convention. The Polymer coding convention is to include the custom element definition in one HTML file [5]. Notably "The CSS styles defined inside the `<dom-module>` are scoped to the element's local DOM.". If CSS styles are imported outside of the template in the `<dom-module>`, they will leak through to other elements in browsers which have not implemented shadow dom yet. Secondly the `Polymer({...})` call is placed inside the `<dom-module>` as it is directly coupled to provided `<template>` definition. Moving the Javscript out of this file makes working with elements significantly harder as reasoning about the code is a lot harder.

SIG states in their e-mail that HTML fragments should be as small as possible to enable reusability. Since custom elements leverage the native platform, there is an extensive usage of HTML to achieve composition and reusability. Template definitions of a custom element describe the structure of the element. Whenever the custom element is used in another custom element, the browser renders the template in the Shadow DOM of the declaration. Template definitions provide by design reusable HTML fragments to reduce the amount of duplicate code. Our manual inspection of the template definitions of our elements shows that the largest template definition consists of 55 lines of code, `create-pull-request`. This is the result of our formatting convention of putting data-bindings of ajax-elements on a new line. Only 8 custom elements are used in this file, which is in our opinion not a problematic amount of declarations.

Moreover, SIG states that JavaScript functions should be split as large functions are less maintainable. The 2 largest functions in our project are `toDateString` in `ToDateStringBehavior` and the inner function of `_parseDiffToGroups` in `pull-request-diff`. `toDateString` is 42 lines of code and was already extracted to a Behavior[6] to enable reusability and is based on the breakpoints set by the Moment[7] library. We considered splitting up this function by creating a separate array which depicts breakpoints, but this would greatly increase the time of understanding how the function works. In our opinion the current solution is the easiest to modify and understand.

The inner function in `_parseDiffToGroups` is 29 lines of code. Separate logic of extracting the fileHeader and the REGEX declaration was already declared outside of the function. The function itself consists of an inner loop with primarily object declarations. Our formatting convention is to declare fields on a separate line, which results in 10 lines of object declaration. Therefore we do not think this function is unmaintainable, as the responsibility of this function is split up in different parts (regex, separate function and the actual looping).

All other functions in our project at most 25 lines of code (which includes white-space and formatting).

The largest file in our project is `github-authenticated-ajax` with 412 lines of code. This element is a wrapper of the `iron-ajax` element built by the Polymer team. At the moment wrappers have to redeclare properties of the wrapped element, and as a result of copying the properties of `iron-ajax` we have 257 lines of property declaration. This is not code written by us, but by an external dependency. To prevent unnecessary copy-pasting, the Polymer team is already working on custom element inheritance which deters the need to copy-paste the properties of the element to be extended. Once this feature lands in the Polymer library, `github-authenticated-ajax` would only be 155 lines of code which we do not consider as too much. We acknowledge the fact

[5]https://www.polymer-project.org/1.0/docs/devguide/quick-tour
[6]https://www.polymer-project.org/1.0/docs/devguide/behaviors
[7]http://momentjs.com/docs/#/displaying/fromnow/

that additional knowledge is required to understand the reason for re-declaring the properties and as stated this should be less verbose once inheritance lands in the Polymer library.

SIG advises to split up the element definitions into separate JavaScript- and CSS-files. As pointed out earlier this contradicts the coding convention of the Polymer library which we follow. SIG already states in the e-mail that this convention is different than the convention of Javascript frameworks. Splitting up JavaScript and CSS never makes it more reusable because this code is always coupled to an HTML element somewhere. The three types of source code form one component that is reusable in itself. Since a custom element declaration is unique for each specific custom element, splitting up the Javascript/CSS from the corresponding HTML template does not provide any benefit regarding reusability. For reusability usecases, behaviors are designed which we use for several Javascript functions we use in multiple custom elements.

In a follow-up e-mail SIG also stated that the usage of private functions is very low and this statistic was a very big factor on our maintainability score. While true private functions do not exist in Javascript, the convention is to prefix the name of a private function with a ”_”. Our calculations show that 80% of the functions defined are prefixed with a ”_”. We are not sure what the source is of this discrepancy between our own calculations and the SIG score.

Concluding the feedback of SIG is focused on the pre-defined convention that is coupled with our library choice. A different coding convention leads to incorrect functioning custom elements as a result of CSS style leaks and developers need significantly more time to reason about the template and corresponding `Polymer({...})` declaration. Therefore we will continue following the coding convention as defined by Polymer to remain with a functional and maintainable product.

7 Product evaluation process

To evaluate the impact that this system can have on code review quality, we designed a quantitative experiment to verify code review performance. The experiment aims to give insight in how the tool affects the reviewers ability to:

- understand changes
- spot flaws
- provide high-quality feedback to the contributor
- timely review proposed changes

The evaluation process is similar to the process employed by Bragdon et al.. Their tool was evaluated by letting developers fix a bug. One group of participants used the developed tool whereas the control group used a standard version of the Eclipse Java IDE.

A challenge when attempting to reliably measure the above things in our system arises from the fact that the effectiveness of our tool leans heavily on how the contributor organizes his/her changes. When measuring reviewer performance in any way, we have to account for the fact that changes in performance may be caused by the way in which code changes were organized. Therefore, measurements will not be representative when we predetermine the ordering used during the experiment.

To mitigate this problem, a possible solution would be to separate participants into three groups: one group where each participant creates an ordering for a predetermined set of changes, one group to perform code review on the changes organize by the previous group, and a control group that gets to review the changes as they would be presented by GitHub.

By having participants of the first group create an ordering we aim to account for possible differences in the quality of such an ordering. If code review performance is affected by our tool in this experiment, it will be quite likely that an ordering of average quality will also affect the code review in a real-world situation.

7.1. Measuring feedback quality

Reviewer feedback quality can be measured on GitHub as well as on our system by analyzing the comments that a reviewer places on the proposed changes. To measure feedback quality, comments should be classified into zero or more of the following (not necessarily mutually exclusive) categories:

- **Question/clarification request:** all comments that contain a question aimed to clarify some aspect of the change to the reviewer.

- **Code style:** all comments that remark upon non-functional aspects of the code like indentation and formatting. While these comments are useful for maintaining consistent code style throughout a repository, they do not add much value in terms of code understanding or defect finding.
- **Implementation details:** all comments that remark upon low-level details in the functional implementation of some code. For example, a comment may advice to use a different function for performance reasons, or a comment may point to a defect on a specific line of code. More comments in this category should increase the probability that a flaw in the code is found.
- **Architectural style:** these are comments that remark upon high-level aspects of the change. A comment may for example suggest a different approach to achieve a solution that fits better in the architecture of the entire system. These comments are the most useful because they increase code understanding for the contributor and can often avoid problems that would arise later on due to bad design. The more high-level comments a pull request receives, the more in-depth the feedback is.
- **Other:** All comments that could not be placed in the above stated categories.

The number of comments is an implicit measure of the amount of code review performed [Gousios et al., 2014]. For large pull requests, a higher number of comments is required to achieve the same review coverage on the proposed changes. Therefore we evaluate the ratio of comments to the number of lines changed as last metric.

7.2. Measuring amount of detected code flaws

The amount of flaws detected in a review can be measured simply as the amount of code comments that identify a flaw as such. However, it is important that the flaws in the change are diverse enough to cover a broad spectrum. We believe the spectrum can range from very low level (e.g.: syntax errors, spelling errors and style inconsistencies) to very high-level (an architectural mistake or design) mistakes.

7.3. Measuring code review time

With each task, the time it takes to review a pull request should also be measured. This gives a high-level indication of the amount of effort it takes to understand proposed changes. Combined with the other metrics discussed, the amount of time it takes to review code with our tool as opposed to GitHub can put the other metrics into perspective. For example: if metrics positively improve with usage of our tool, but the review time drastically increases, this might indicate that there is a heavy downside or even a design flaw in the tool itself. The reverse is also true, a faster review process does not mean better review quality.

Table 7.1: Measurements obtained during the experiment

Name	Description	Unit of measurement
Clarification	Comment with a clarifications request about an implementation detail	Number of comments
Code style	Comment about style, including but not limited to indentation and naming	Number of comments
Implementation	Comment about a low-level implementation of certain functionality	Number of comments
Architecture	Comment about high-level architecture of combined proposed changes	Number of comments
Comments per LOC	The total amount of comments posted relative to the LOC	Number of comments divided by LOC
Code review time	The amount of time spent to employ the complete code review	Minutes

7.4. Experiment Protocol

Purpose The purpose of the experiment is to test the effectiveness of the developed tool in terms of improved code understanding and quality of a code review.

Materials

1. Laptop with internet connection
2. Video camera

Methods

1. Determine the changes to be reviewed by the participants. The changes must contain several flaws/bugs and questionable design decisions for reviewers to comment on.
2. Introduce all developers to a small code base and receives a short introduction to our tool.
3. Split up all developers into three groups.
4. Explain the first group of participants what kind of a change was made to the codebase.
5. Instruct the first group to create a pull request and ordering for this change.
6. Randomly distribute the generated orderings over all participants in the second group.
7. Instruct the second group to code review the provided ordering using the developed tool. Instruct the third group to code review using the regular GitHub interface. All metrics described in table 7.1 must be gathered during this stage. Record the activity of the code reviewer with the video camera, positioned diagonally behind the code reviewer.

Control treatment The determined changes remain unordered and are reviewed using the regular GitHub interface by the third group.

Data interpretation For each stated metric, a graph will be used to compare the metrics of the research group with the control group. On the x axis the second and third group are plotted. On the y axis is the average of the unit of measurement plotted of each group.

7.5. Threats to validity

The problem of a predetermined ordering was avoided by having participants order code changes. While this accounts for one free variable, there is another important variable that could not easily by taken into account in this experiment. The bias from a predetermined ordering was mitigated, but there is still an implicit bias in the code changes used in the experiment itself. We choose to set the changes as well as the code base of these changes to control the extent to which reviewers can comment on the changes. A too trivial pull request would not see the benefit of our tool as there would not be enough to review. A too large pull request would confuse participants as they are not very familiar with the code. Researchers executing the experiment have to strike a balance in choosing a pull request by size and complexity. This may introduce a bias because the developed tool may work better on some pull requests than on others. A possible way to avoid this is to design different experiment that tests a larger amount of reviews on real-world pull requests.

8
Conclusion

During the project we have developed PReview-Code to improve the code review interface. Preview-Code allows code contributors to present their contributions in a more dynamic and logical way.

We started started the project with a research of all recent advancements in the field of software engineering regarding code review. In the two weeks of initial research we have discovered and designed several improvements to the code review process. The improvements were translated into requirements of PReview-Code.

In the following seven weeks PReview-Code was developed. We started by building a prototype used to assess the usability of the system with the RITE method. Several users have used the prototype and provided feedback to improve several features of PReview-Code.

The two main features of PReview-Code are ordering of changes and a visualization of the structure of the changes. Code contributors can now highlight important changes by displaying them on top as well as moving less important changes down. These capabilities can provide significant help for reviewers to understand the contributions and review all proposed changes.

In the 2D space logical connections can be created between the groups to also assist the code reviewer with understanding the overall structure of the changes.

Taking all feedback into account, we iteratively improved PReview-Code and implemented several new features. The full system has to be evaluated to assess an increase of the quality of code reviews as a result of an improved code review process. Therefore we have designed an experiment which researcher can execute aimed at the verification of the effectiveness of the product.

While the product could be used as-is, we recommend further development to polish the tool and incorporate the remaining requirements.

9 Discussion

In this section a reflection is given on multiple aspects of the project and product.

9.1. Reflection on requirements

The requirements we defined did not include the requirement to achieve feature parity with GitHub. In the RITE sessions the majority of feedback was targeted towards missing features the participant was used to have working with GitHub. As a result, a lot of time was spent on achieving feature parity and fine-tuning/improving the existing features. This was not integrated into our must-haves, which it should have been.

Our existing must- and should-haves were delayed as we did not initially calculate the time spent on processing the feedback. In the halfway meeting with our client we discussed this issue and agreed that the first priority was to implement all must-haves and postpone the should-haves. Notably the should-have Bitbucket integration was delayed. This should-have was initially designated for a company contact of our client that wanted to test our software.

Secondly, some features were delayed including comment hierarchy and recommend reviewer. The comments were initially planned to be implemented in week 6, but were delayed to week 8. Storing the comment hierarchy required a database which had to be investigated. More information about our database is available in section 9.2. The recommend reviewers has been postponed for the time being, as there is ongoing research by other researchers on this topic. Notably a Bachelor intern of our client is performing research on appointing the best reviewer for a given pull request.

What we learnt

Concluding we have realized that some requirements may not be apparent when initially stating the requirements. This is especially noticeable when requirements arise from user testing. In a future project a dynamic requirement must be included to account for potential crucial features that alpha/beta users report.

9.2. Firebase integration

Initially we wanted to build the database with Hibernate[1] on the Java backend. However, this implementation would have several major challenges accompanied with it. First of all the setup required us to maintain the database, services and corresponding REST end-points. Secondly we wanted our application to be updated in real-time, which would require us to program some sort of live update system such as websockets. Given the time-constraints and the technical challenges, we looked into ready-to-use alternatives.

Our solution was to integrate Firebase into our application, as explained in 5.1.4. This turned out to be a very good decision because the flexibility and simplicity of Firebase allowed us to spend

[1] http://hibernate.org/

much less time writing back-end code and much more time on the front-end of the application, which is critical to the success of this tool.

What we learnt

All in all we did not anticipate on using a ready-made solution such as Firebase and did not include this into the requirements. The extra external dependency was discussed later with our client in the half-way meeting where we agreed to use Firebase instead of building/managing our own database solution. We think that discussing potential technologies in a meeting with the client is a sufficient solution to revert earlier stated requirements.

9.3. Reflection on project process

The project was split into two different parts: the research phase and the development phase. The research phase lasted the first two weeks of the project, all later weeks were designated to the development of our product.

Research phase

During the research phase, our primary task was to read papers on modern code review. The papers were supplied by our client or chosen by ourselves. All papers supplied by our client were read by all team members. After reading a paper, we had a quick discussion regarding the content of paper. In this discussion we wrote a quick summary and a list of (in our opinion) most important scientific contributions and key points applicable for our own product. The paper that were chosen by ourselves were first read by one team member to assess the quality of the paper. If the content was deemed important for our product, all other team members also read the paper and discussed as described above.

In the research phase we did not employ the SCRUM methodology just yet. Instead, we wrote down the paper we were reading and kept a central document up-to-date with the extracted scientific contributions for each paper.

Development phase

The development started with building the prototype for the frontend. Our choice of prototyping tool is described in section 4.2. Prototyping and building code for our actual application resulted in faster development in the long-term, but it took us longer to start the RITE evaluation. Originally it was planned to start the RITE evaluation in week 3, however due to vastness of user-tested features this was postponed to week 5. In week 5 we had a sufficiently working prototype to let the first few participants use the system. Since the product development was in its early stages, we were able to adopt the feedback from the participants very quickly. Moreover since we were building the actual application as prototype, we were able to progress faster later on as all evaluated major features were user tested thoroughly. It was also decided that the RITE evaluation would not last just one week, but it would be performed throughout the project. Therefore we had a full evaluation cycle at every significant point in our development process. This was a successful strategy, as our main task was to improve the user interface. By adopting user feedback as early as possible, we were able to adjust our product much faster.

After we finished the first RITE evaluation, we started planning the other planned major features of our product. At this point we started to employ the SCRUM methodology as everyone started to work on separate subsystems (up to this point everyone was working in the same subsystem). Using SCRUM storypoints (one storypoint is worth roughly 2 hours of development time) we assigned tasks to every team member until everyone had 15 storypoints worth of tasks assigned to him/her. 15 points per week amounts for 30 hours of development time per person. This left roughly 10 hours per week for inaccuracies in our time estimations and non-development tasks including code reviews, meetings with our client, team discussions and dealing with unexpected challenges.

What we learnt

In total the development phase lasted 7 weeks, since the deadline for the project was set in week 9 and we performed 2 weeks of research. 7 weeks proved to be too short to implement all features we wanted to implement. As explained in section 9.1 we initially did not include the requirement of feature parity with GitHub. Therefore we not only had a limited amount of time of all major features we wanted to implement, we also had to implement existing GitHub features at the same time. We will use the weeks up to the demo to polish existing features.

For future projects we should probably either reduce the number of must-haves or discuss the deadline. In this project we do not have influence on the deadline, so a reduced number of must-haves would be the only option. This should of course be discussed extensively with the client to meet their expectations.

10 Recommendations

In this section some recommendation are done when continuing this project. These recommendations are given on multiple aspects, namely the experiment, development and running of the tool.

10.1. Experiment

The experiment outlined in chapter 5 should provide for a good experiment setup for verifying tool effectiveness. Sadly, due to lack of time, we were not able to execute this experiment during the project ourselves. Evaluating the tools performance would be a recommended step in future work.

10.2. Development

As discussed in the previous chapter, some requirements are still left to be implemented. Working on these should be top-priority in any next development effort. Alongside the remaining requirements, some improvements can be made to the styling and general user experience. All these possible improvements are documented in the backlog on our public scrum-board. These backlog items can be accessed through Waffle[1] or GitHub[2]. Furthermore, an improvement to the design could be made in order to motivate contributors to order their changes properly. Currently, the system leans heavily on the quality of the ordering produced by contributors. If contributors refuse to order their changes, the main benefit of using this tool is gone. A possible strategy would be to ease the ordering for contributors by automatically suggesting an ordering, however, the design space is not limited to this sole option. More options should be evaluated in the future.

10.3. Architecture

Architecturally, the system was designed to allow its developers to focus mainly on front-end user experience. This is the main reason we choose Firebase as a means of storing data, because it reduces the amount of code needed in the back-end. However, while Firebase is great for simple applications, it gives little control over when and how data can be accessed by users. This implies that whenever access control must be enforced, it must be handled by a separate back-end server. When the application grows, so will the back-end. At some point, the amount of responsibilities handled by the back-end will become large enough that there is no additional gain in out-sourcing data storage to Firebase. Another thing to take into account is that Firebase is a closed external dependency. Many subtle design decisions made by the developers of Firebase are initially not obvious when using the service, but will eventually leak through into the Preview-Code code base. This makes it hard to anticipate how design decisions made by Firebase will affect our application when it grows. For example, it is unknown how exactly transactions are

[1] https://waffle.io/preview-code/rite-evaluation
[2] https://github.com/preview-code/rite-evaluation/issues

handled and what guarantees about atomicity are provided. This could cause problems when Preview-Code reads and writes to Firebase many times concurrently. Furthermore, it will be a significant challenge to resolve incompatibilities between our architecture and Firebase when such problems do come up, since we have no influence on how Firebase is built or how it behaves. Therefore, when Preview-Code must be deployed on a large scale, we strongly advice to consider alternatives for data storage.

10.4. Running the tool

Because the developed product consists of a web application, there are special requirements for running the system. First of all, as described in chapter 6 the application contains a light-weight back-end component. This component has to be run on a server with access to the Java API and an internet connection. Secondly, the front-end component has to be made available on a web-server as well. This server should just be capable of serving HTML-files. When both the front- and back-end are available on one (or two) servers, users can browse to the configured domain and url to start using the application. It is recommended that when one wants to publish the application for global use, that the front-end files are served through a content distribution network (CDN). This enables fast loading times of the application.

A
Project description

Code review is a commonly used process to ensure the software quality of a product is maintained at a high level. One of the popular sites at the moment to employ code review is GitHub. GitHub provides a software repository hosting service and also has an integrated code review interface.

The current GitHub interface is not suitable for reviewing large pull requests. It does not scale well and offers no dynamic possibilities to highlight important parts of the pull request and suppress changes with minor relevancy.

Therefore a tool must be developed to display code changes in a more dynamic and intuitive way. Authors must be provided with a simple yet effective way to explain their changes. As a result the reviewer of the pull request will have an easier process reviewing, understanding and providing feedback on the pull request.

Students must first employ research on the state-of-the-art science regarding code review and digest pain-points of the current code review process. Using the obtained knowledge, the tool must be developed which takes into account the available research. This tool must be evaluated with usability tests to ensure the goal of the tool is achieved. Later an experiment must be designed to verify the effectiveness of the improved user interface.

B Infosheet

Project title: Enhanced GitHub code review
Client: TU Delft
Presentaion date: 01 July 2016

Code review is a commonly used process to ensure the software quality of a product is maintained at a high level. One of the popular sites at the moment to employ code review is GitHub. GitHub provides a software repository hosting service and also has an integrated code review interface. The current GitHub interface is not suitable for reviewing large pull requests. It does not scale well and offers no dynamic possibilities to highlight important parts of the pull request and suppress changes with minor relevance.

Challenge: Improve the quality of code reviews by offering an enhanced revision management system.

Research: Recent research in the area of software engineering has provided some interesting insights about the challenges of code reviews. For example, the main benefit as well challenge in reviewing code lies in *code understanding*. Teams performing reviews experience an increase in team awareness and general understanding of the code they're working on. However, performing a good code review hinges on understanding the proposed changes. Our tool focuses on improving code understanding.

Process: After a two week literature study, we started the development and usability-evaluation by means of an iterative approach. After each week, new features and changes to the product were evaluated from a usability perspective with the RITE (Rapid Iterative Testing and Evaluation) method.

Product: A tool was developed to display code changes in a more dynamic and intuitive way. The resulting product consists of a web application with light-weight server back-end. The app communicates with the Github API to provide in interface in which a user can create and review pull requests.

Outlook: The current version of the project could be considered a Beta, which means the basis of the product is usable. There are still features left that could be implemented or improved. From a research perspective, we designed an experiment aimed at the verification of the effectiveness of the product.

Team

Eva Anker: Interested in robotics and cyber security. Responsible for the backend and firebase integration.
Tim van der Lippe: Software engineering and open source enthusiast. Responsible for application structure and dataflow.
Thomas Smith: Interested in functional programming and software architecture. Responsible for the application UX and style.

All members contributed to implementing features, writing chapters in the report and preparing the final presentation.

Client: Alberto Bachelli (sback.tud@gmail.com): TU Delft - EEMCS - SERG

The final report for this project can be found at: http://repository.tudelft.nl

C

Plan of action

In this chapter the plan of action is shown. This plan shows the major features that will be worked on in a particular week. However, more features could be implemented due to the difficulty in time allocation of features.

Week	1	2	3	4	5	6	7	8	9	10
Problem definition	🟦									
Problem analysis	🟦									
Research plan	🟦	🟦								
Requirements		🟦								
Design with RITE		🟦	🟦							
UI development			🟦							
Basic application (login and pull request overview)				🟦						
Pull request view (groups)					🟦	🟦				
Comments						🟦				
Status							🟦			
Recommend reviewer							🟦			
Workload indicator								🟦		
Report								🟦	🟦	

Table C.1: Plan of action

D
E-mail first SIG feedback

[Analyse]

De code van het systeem scoort 2 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code benedgemiddeld onderhoudbaar is.

Het is opvallend dat het systeem bestaat in grotendeels van alleen HTML code. Zowel CSS als Javascript code wordt in de HTML bestanden geschreven. Dit maakt de code slecht leesbaar en lastig te onderhouden. De HTML bestanden zijn lang, wat leidt tot lage score van Unit Size.

Om de onderhoudbaarheid van het systeem te verbeteren is het aan te raden om de structuur van de code te verbeteren. Begin met het verplaatsen can de CSS code en de JavaScript code naar aparte bestanden. Denk aan modulariteit in de code. De html bestanden moeten klein zijn, zodat een HTML fragment zou kunnen worden hergebruikt. Dezelfde geldt voor de Javascript functies. Het opsplitsen van de lange Javascript methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. We weten dat dit advies afwijkt van de manier waarop Polymer het in hun eigen tutorials aangeeft, maar dit is een punt waarop het advies van één framework afwijkt tegen de heersende mening.

Verder is het goed om te zien dat jullie testcode hebben geschreven. De verhouding tussen de hoeveelheid test-code en de hoeveelheid code voor productie is op dit moment redelijk goed, 1:2. Hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Bibliography

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464. ACM, 2010a.

Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512. ACM, 2010b.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014. doi: 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

Georgios Gousios, Margaret-Anne Storey, Alberto Bacchelli, Laura MacLeod, Andreas Bergen, Jian Lü, David S Rosenblum, Tevfik Bultan, Valerie Issarny, Schahram Dustdar, et al. Work practices and challenges in pull-based development: The contributor's perspective. *IEEE Software*, 32(1), 2015a.

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 of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015b.

Bruce Hanington and Bella Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

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*, pages 192–201. ACM, 2014.

Michael C Medlock, Dennis Wixon, Mark Terrano, R Romero, and Bill Fulton. Using the rite method to improve products: A definition and a case study. *Usability Professionals Association*, 51, 2002.