Octopull: integrating Static Analysis with Code Reviews

Version of December 14, 2015

Reinier M. Hartog

Octopull: integrating Static Analysis with Code Reviews

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Reinier M. Hartog
born in Voorburg, the Netherlands

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

©2015 Reinier M. Hartog. All rights reserved.

Octopull: integrating Static Analysis with Code Reviews

Author: Reinier M. Hartog
Student id: 4090802
Email: r.m.hartog@hotmail.com

Abstract

Teams using modern day software engineering practices often incorporate code reviews as a quality assurance step in their development. These code reviews are intended to uncover software quality defects before code changes are incorporated into the project.

Certain classes of these software quality defects can be detected by so-called static analysis tools. These tools have seen increasing uptake and are found to be effective at finding relevant quality defects in these classes.

Several tools to integrate the static analysis results into code review have been created, such as SCRUB and Review Bot. However, these tools were created specifically for internally used platforms and their source is not made available. In this thesis, we propose and implement a tool called Octopull, which is an open-source implementation that incorporates static analysis results in the user-interface of the GitHub platform.

We evaluate the tool by performing a user-study on undergraduate students. Our study shows no significant effect of using the tool on the effectiveness of their code review sessions.

Thesis Committee:

Chair: Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor: Dr. A. Bacchelli, Faculty EEMCS, TU Delft
Committee Member: Dr. J.A. Redi, Faculty EEMCS, TU Delft

Preface

This thesis was written as part of my Master's Thesis project at the Delft University of Technology. The basis for this document is my work done during the period from September 2014 to December 2015.

In my thesis, I present an open-source tool that integrates static analysis with code review on GitHub. Using this tool, I conduct a user-study on undergraduate students to determine if the tool is effective.

The creation of this thesis has been both a challenge and a great learning experience. I would like to thank Dr. Alberto Bacchelli, for his supervision and support, as well as Dr. Zaidman and Dr. Redi for agreeing to be part of the Thesis Committee. I also want to thank my girlfriend, my friends and family for bearing with me, supporting and encouraging me.

Reinier M. Hartog
Delft, the Netherlands
December 14, 2015

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Research Questions . . . 1
- 1.2 Outline . . . 2

2 Background and Related Work 3
- 2.1 Testing . . . 4
- 2.2 Pull-Based Development . . . 7
- 2.3 Travis CI for GitHub . . . 9
- 2.4 Integrating Static Analysis and Code Review . . . 9

3 Approach 11
- 3.1 Design of the Experiment . . . 11
- 3.2 Hypotheses Formulation . . . 18
- 3.3 Measured Variables . . . 19
- 3.4 Data Collection . . . 21
- 3.5 Data Analysis . . . 22

4 Implementation 25
- 4.1 Architecture . . . 25
- 4.2 Travis configuration . . . 27
- 4.3 Octopull Server . . . 29
- 4.4 Octopull Plugin . . . 32
- 4.5 Data retrieval . . . 34

5 Results 37

5.1 Overview of the collected data . . . 37
5.2 Developer interest and usage . . . 38
5.3 Threats to validity . . . 43
5.4 Discussion . . . 45

6 Conclusions and Future Work **47**
6.1 Contributions . . . 47
6.2 Future Research . . . 48

Bibliography **49**

A Rubrics **55**
A.1 Working Version Rubric . . . 55
A.2 Final Version Rubric . . . 56
A.3 Quality Rubric . . . 56
A.4 Sprint Management Rubric . . . 57

B Pretest Questionnaire **59**

C Posttest Questionnaire **63**

List of Figures

2.1 The conversation view of a Pull Request on GitHub. 8
2.2 The files view of a Pull Request on GitHub. 9

3.1 The design of the experiment in the notation of Shadish et al. [54] 11
3.2 An example of the Octopull tool warnings. 15

4.1 A high-level overview of the implemented solution. 26
4.2 Travis CI switches for each of the user's projects. 27
4.3 An example Travis configuration file for a project using Octopull. 27
4.4 The underlying Git structure of a Pull Request. 28
4.5 A high-level overview of the architecture of the server project. 30
4.6 An overview of the interactions that occur when a Travis build completes. . . 30
4.7 An overview of the interactions that occur when an Octopull users browses GitHub. 31
4.8 The Octopull plugin prompts the user to log in to the server. 33
4.9 The Octopull plugin provides a filterable overview of the warnings for this Pull Request. 34
4.10 The Octopull plugin displays the warnings in the 'diff view'. 34
4.11 A comment generated from an Octopull warning in the 'conversation view' (*top*) and in the 'files view' (*bottom*). 35

5.1 Boxplots of the absolute difference in violations before and after the pull request. 40
5.2 Boxplots of the fractional difference in violations before and after the pull request. 40
5.3 Code review priorities from pretest to posttest. 42
5.4 Contingency tables of Octopull usage versus Professional Software Engineering experience (*left*) and Octopull usage versus Open Source Contribution (*right*) 43

5.5 Bar-graphs showing the percentage of students that indicated a certain experience level for each tool, split by Octopull and non-Octopull users. 44

Chapter 1

Introduction

Over the last decades, software has come to play an increasingly large role in our lives. Many of our daily activities are supported by software systems and a lot of industries are becoming more dependent on them. As the impact of these software systems grows, so does the impact of possible defects within them, and with that, the cost.

An important factor in reducing the cost of software defects is to detect them early and prevent them from being released. Testing has widely been adopted in the industry for this purpose [19]. Besides the detection of defects, testing can have additional benefits, such as better understanding of the product and demonstrating that the product meets the users' expectations [37].

In interviews by Bacchelli and Bird [13], the most stated reason for performing code reviews is finding defects. However, in their survey of review comments, they found that comments related to defects are few and often address small or superficial problems. Instead, the largest group of comments is code improvement, in which issues such as formatting, naming and small inefficiencies are addressed.

Bacchelli and Bird note that tools already exist to detect many of these issues automatically, so that reviewers can focus on defects that are harder to detect. The purpose of this thesis is to investigate whether it is in fact useful to further integrate these tools with code reviews.

In the rest of this chapter we formulate the main question of this thesis and split it into smaller research questions. Finally, an outline of the following chapters is given.

1.1 Research Questions

We formulate the main question of this thesis as follows:

"Does integrating Static Analysis with Code Review increase the effectiveness of the Code Review process?"

To answer this broad question objectively, we need to define what we mean by the effectiveness of the code review process. To define this term, we need to define what outcome we desire from code reviews.

Bacchelli and Bird [13] identified several motivations for developers to perform code reviews, such as: 'finding defects', 'code improvement', 'finding alternative solutions' and 'knowledge transfer'. In this thesis report, we limit ourselves to 'finding defects' and 'code improvement'. These were identified as the two most common motivations for code review and overlap with the functionality that most static analysis tools provide. For instance, FindBugs and PMD are static analysis tools that detect patterns that are commonly buggy, and CheckStyle is a static analysis tool that indicates code quality issues such as method lengths and naming concerns.

Given these desired outcomes, we define the effectiveness of the code review process as the amount of defects and code improvement opportunities found and addressed in a code review session, where more addressed findings indicate a higher effectiveness of the review. We split our main question into smaller research questions to be answered seperately.

Research Question 1. *Are developers willing to use a tool that integrates warnings into their code reviews? If they do, how do they use and experience the tool?*

Research Question 2. *What is the effect of displaying warnings on the amount of resolved warnings during code review?*

Research Question 3. *What is the effect of displaying warnings on the discussion of code improvements and other concerns during code review?*

Research Question 4. *What is the effect of displaying warnings on the developers' perceived priorities during code review?*

1.2 Outline

The rest of this thesis is outlined as follows. Chapter 2 goes deeper into the background of the problem and summarizes related work. In Chapter 3, the setup of the study and the experiments is explained. Chapter 4 elaborates on the design and implementation of a tool, Octopull, that is used for the experiments. Chapter 5 presents and discusses the results of these experiments. Finally, Chapter 6 summarizes the conclusions to be drawn from the results, and discusses possibilities for further research.

Chapter 2

Background and Related Work

In this day and age, software is all around us. Software systems control the appliances in our homes, the traffic lights, our work computer and machinery. With the growing impact of software on our lives, the impact of software failures grows as well. Most people have at some point witnessed software failures. Such failures can cause annoyance, for example when an application on your smartphone is failing to respond. They can also cause a major disruption, such as when a software failure causes a power outage. The consequences of such failures can also be fatal. For example, the Therac-25 machine, designed for radiation therapy, administered a severe overdose of radiation to six patients because of a software failure [44].

Clearly, the impact of these software failures differ, yet most (if not all) of them have some negative and undesirable effect. It makes sense that some investment is made into preventing them. The first point of preventing software defects, is to prevent the mistakes from ever being made. By educating developers on good coding practices, and from experience with previous defects, developers will learn to avoid common mistakes.

Because people are fallible and make mistakes, even when experienced, the practice of pair programming has been proposed. This practice, where two developers synchronously collaborate on the same piece of code, has been found to reduce the number of defects introduced [23]. This can be seen as the second point of prevention, where software defects are found and removed while the code is being written.

The third point of prevention is when a piece of code has been written. At this point, the code can be *tested* for defects. Here, testing is defined very broadly, as a procedure that verifies that a piece of code meets certain requirements. Note that, even though testing applies to a piece of code *after* it has been written, this does not necessarily imply that the code has to be considered done. For example, the practice of Test-Driven Development (TDD) [16] suggests that a test should be created first. After creating the test, code should be written in small steps and repeatedly subjected to the test, until the test passes.

In this chapter, we further explore the topic of testing. Thereafter, we discuss the pull-based development model. We discuss the Travis CI service for GitHub and finally, we consider several integrations between static analysis tools and code review.

	Dynamic	**Static**
Manual	User-based testing	Code Inspection Code Review
Automatic	Unit testing Automated acceptance testing	Static analysis tools

Table 2.1: A taxonomy of testing techniques, divided by dynamic vs. static and manual vs. automatic
The typical types of testing in each quadrant are shown.

2.1 Testing

Testing can be subdivided into static and dynamic testing. Dynamic testing refers to testing where the code is executed with certain inputs, and the output is verified against the output that the requirements of the test dictate. Static testing, on the other hand, refers to verifying requirements by analysing the code itself, without executing it [37]. Testing can also be subdivided into manual and automatic testing. Manual testing refers to a procedure performed by a person, to assess whether code meets the requirements, whereas automatic testing refers to a procedure performed by a tool. Combining these two dimensions yields the taxonomy as seen in Table 2.1. Each type of testing will be discussed in the following sections.

2.1.1 User-based testing

User-based testing refers to testing with individuals that represent the end-user population [58]. It is a form of dynamic testing, as the software is being executed, and it is a manual process, as the interactions with the software are performed by the users. User-based testing can be subdivided in exploratory testing, comparison testing and validation testing [58].

The purpose of exploratory testing is to characterise how users interact with a system and/or how they would like to [58]. This allows the design of the software to be adjusted towards the end-users needs.

Comparison testing is similar to exploratory testing, but is used to identify the most appropriate interaction pattern from a fixed set of alternatives [58].

Validation testing is used to validate the usability of the product and to determine if the product is fit for its purpose. It is used to identify problems and defects in the software while in use [58].

User-based testing is primarily aimed at detecting problems and defects with the usability of the product [58]. While this is useful for detecting conceptual problems and problems with the user interface, it is less likely to uncover lower level defects in the code.

2.1.2 Unit and Automated Acceptance testing

Unit testing refers to the automated testing of an individual unit of code [58]. It is realised by writing an additional piece of code, the unit test, which provides the code to be tested with input and verifies its output and/or state. Similarly, an automated acceptance test [58] is an automated version of the acceptance script for a certain feature. Instead of a single unit, an acceptance test targets all parts of the system that together provide the specific feature being tested.

Both forms of testing are a type of automatic, dynamic testing, where the test results can be obtained with little or no human intervention. They are dynamic because they run (part of) the system under test. These types of tests are very suitable for repetitive tasks, which humans generally perform slowly and in an error-prone fashion. Because of the reliable and repeatable nature of this type of testing, it has become particularly popular [58].

2.1.3 Code Inspection and Code Review

Code Inspection or Code Review are the practices in which humans evaluate the source code or other artifacts of the software development process [58]. As such, it is a manual and static testing technique, that relies on human judgement.

In 1976, Michael Fagan published a formalized inspection process for code [31]. Each phase of the process is thoroughly documented. Arguably the most important phase of the process is a group meeting, in which generally four people in different roles come together to discuss and review the code. A decade later, Fagan published the lessons learned from applying this practice over the years [32]. He argued that, through this process, between 60 and 90 percent of all defects were found.

Since the introduction of Fagan's inspection process, many alternatives have been proposed, such as Two-Person Reviews [20], Verification-Based Inspection [29], Phased Inspections [43], and N-fold Inspection [47]. What all these approaches have in common, is the face-to-face review meeting.

In contrast to the other approaches, Parnas and Weiss [51] suggested an approach called Active Design Review, which does not require meetings. Votta [57] found that scheduling of the meeting alone cost 20% of the review interval. Prompted by these findings, Johnson and Tjahjono [40] compared meeting-based reviews to non-meeting-based reviews. They found meeting-based reviews to be more costly, but not more effective than reviews without a meeting.

Surveying the current state of the practice in 2013, Bacchelli and Bird used the term Modern Code Review [13] to refer to the lightweight approach that is currently being adopted by many organisations and open source software projects. In contrast to Fagan's process and its derivatives, it is characterized by an informal and tool-based approach.

Bacchelli and Bird looked at the expectations and outcomes of code review at Microsoft. There, they use a collaborative code review tool called CodeFlow [13], which is similar to Google's Mondrian [42], Facebook's Phabricator [33] and open-source Gerrit [4]. Using these tools, a developer can submit their changes for review. Potential

reviewers are notified and can asynchronously inspect the code and comment on it. Furthermore, in interviews with developers and managers, the motivation for code review was investigated. The most important reason was found to be 'finding defects', with 'code improvement' in second place. Changes to the code are considered code improvements if they do not involve correctness or defects of the code. Examples include readability and layout changes, adding comments or removing dead code. A categorization of review comments showed, however, that comments regarding code improvements were most prevalent. Comments about defects were the fourth most common category out of nine, accounting for only 14% of all comments. Furthermore, review comments about defects were found to mostly address small or superficial concerns.

Tools exist to detect many of the 'code improvement' concerns, as well as many of the smaller defects uncovered in these reviews. These static analysis tools are discussed in the next section.

2.1.4 Static Analysis Tools

Static analysis tools have a similar purpose to code review or inspection: to detect problems and defects in software, without executing it. Given the popularity of dynamic automated testing, it seems natural to automate this task as well.

While static analysis is not capable of showing that the code is functionally correct, it can be used to detect common defects, or produce information useful in further manual code review [58].

There are two main categories of analyses that static analysis tools can perform: code analysis and program analysis. Code analysis refers to analysis of the code through parsing and pattern matching, but does not take into account the execution logic of the program. Because of this, this type of analysis is most suited to detect simple patterns, enforce a certain code style and address readability issues. Examples include warnings for lines that are too long or classes that do not have a descriptive comment. Program analysis refers to the analysis of properties of a program, through the use of methods like data flow analysis, control flow analysis and symbolic execution. This type of analysis can be used to detect common causes of defects, such as unitialized variables or race conditions.

In this thesis report, we will discuss three static analysis tools for Java, namely CheckStyle [1], PMD [6] and FindBugs [25]. In Table 2.2, a categorisation of these tools is given: the target column indicates whether the analysis is performed on the source code, or the compiled byte code, the code analysis and program analysis columns indicate the type of analysis that this tool provides.

Static analysis tools can aid in the early detection of defects and maintainability issues [48]. PMD and FindBugs have been shown to provide useful feedback on possible defects [11, 12], while CheckStyle is more suitable for checking coding conventions [55].

It seems natural to combine static analysis tools with code reviews. As Bacchelli and Bird noted [13], many code review comments are concerned with code improvement and small defects, which can be detected by these tools. They suggest that by automating the detection of these problems, reviewers can focus on more in-depth issues.

Tool	Target	Code Analysis	Program Analysis
CheckStyle	Source code	yes	no
PMD	Source code	yes	yes
FindBugs	Byte code	yes	yes

Table 2.2: A categorisation of the static analysis tools considered in this thesis.

2.2 Pull-Based Development

Since the advent of distributed version control systems (DVCS), such as Git [56], the pull-based software development model emerged as a paradigm for distributed software development [36]. The pull-based model is interesting in the context of this work, as it incorporates code reviews in the development workflow.

Using the Pull-Based development model is a natural fit for open-source projects [36]. A *core team* manages a repository of the source code, to which other developers have *read-only* access. Developers can create their own copy of the repository, called a fork, in which they can apply their changes. To contribute the changes to the original repository, they can issue a Pull Request, so that the core team can make the accept-or-reject decision.

The pull-based development model, as popularized by services like GitHub and BitBucket, is based around such Pull Requests (PRs). A developer can issue a Pull Request against a specific branch in a repository, indicating a set of changes they would like to see applied there. The changes can then be reviewed, and an accept or reject decision made. If the decision is made to accept the changes, they are merged into the target branch.

Pull Requests can also be used between branches of a single repository, in a model that is known as the Shared Repository [36]. While this is not a technical necessity (developers have the ability to apply their changes to the target branch directly), it can be used as a mechanism to review the changes and foster discussion.

Gousios *et al.* [36] found that the pull-based development model increases the awareness of the developers, allows for more community engagement and makes it possible to incorporate new contributions faster.

In this work, we will limit our scope to Pull-Based development on GitHub. The next section describes how GitHub's user-interface supports this development model.

2.2.1 Pull-Based Development on GitHub

GitHub has strong support for Pull Requests. In fact, GitHub uses them heavily for their own internal development[1]. They have built a user interface around Pull Requests to

[1] https://github.com/blog/1124-how-we-use-pull-requests-to-build-github

Access java.lang.management.* via reflection #1187

Open rohitagr wants to merge 1 commit into junit-team:master from rohitagr:reflectivemanagement

Conversation 3 Commits 1 Files changed 9

rohitagr commented on 7 Aug

Several platforms, including Android and j2objc, do not provide the java management APIs and crash or otherwise fail when encountering code that imports or uses these APIs directly.

Access java.lang.management.* via reflection ... 61d7031

kcooney commented on 7 Aug Collaborator

Thanks for doing this. I'm surprised no one reported previously that these APIs don't work on Android.

The changes LGTM

@marcphilipp any concerns?

All checks have passed
1 successful check
Show all checks

This branch is up-to-date with the base branch
Only those with write access to this repository can merge pull requests.

Figure 2.1: The conversation view of a Pull Request on GitHub.

foster discussion: users can post comments on the 'main' discussion of the pull request, as well as on specific changes in the code. Furthermore, all events that occur on a Pull Request are shown in a timeline, giving a clear overview of the changes that are happening, which raises developer awareness [36].

The 'conversation view' of an example Pull Request on GitHub is shown in Figure 2.1. At the top of the page, the Pull Request title, number, base branch and head branch are shown. The base branch points to the place where the changes will be merged, the head branch points at the newest version with all changes. The conversation timeline shows comment messages and other relevant events, such as commits, in chronological order. At the bottom of the page, the current status of the Pull Request is shown, as reported by tools external to GitHub.

The 'files view' of this same Pull Request is shown in Figure 2.2. In this view, each changed file is shown in a so-called 'diff view'. This view is related to the `diff` utility [2], and shows the difference between the initial version and current version of the file.

Access java.lang.management.* via reflection #1187

Open rohitagr wants to merge 1 commit into junit-team:master from rohitagr:reflectivemanagement

Conversation 3 Commits 1 Files changed 9 +319 −4

Showing 9 changed files with 319 additions and 4 deletions. Unified Split

4 src/main/java/org/junit/internal/runners/statements/FailOnTimeout.java View

```
@@ -1,7 +1,5 @@
 1  1   package org.junit.internal.runners.statements;
 2  2
 3     -import java.lang.management.ManagementFactory;
 4     -import java.lang.management.ThreadMXBean;
 5  3   import java.util.Arrays;
 6  4   import java.util.Collections;
 7  5   import java.util.List;
@@ -12,6 +10,8 @@
12 10   import java.util.concurrent.TimeUnit;
13 11   import java.util.concurrent.TimeoutException;
14 12
   13  +import org.junit.internal.management.ManagementFactory;
   14  +import org.junit.internal.management.ThreadMXBean;
15 15   import org.junit.runners.model.MultipleFailureException;
16 16   import org.junit.runners.model.Statement;
17 17   import org.junit.runners.model.TestTimedOutException;
```

Figure 2.2: The files view of a Pull Request on GitHub.

2.3 Travis CI for GitHub

Travis CI is a Continuous Integration [35] service specifically created for GitHub. It builds every change to the mainline (usually the `master` branch) and runs the automated tests, to ensure that new changes integrate with cleanly. Additionally, it builds and tests the merge commit for every pull request on GitHub, that is, the commit as if the pull request were accepted. The status of this process can be seen on the website of Travis CI, but is also reported on GitHub at the bottom of the 'conversation view' of a pull request, as we saw in Figure 2.1. The service is available for free for public repositories on GitHub.

2.4 Integrating Static Analysis and Code Review

Several studies into the integration of static analysis and code review have been performed. In this section, we summarize their findings.

Source Code Review User Browser (SCRUB) [39] is a proprietary tool developed at NASA's Jet Propulsion Laboratory (JPL), that combines code review with static analysis results to make the code review process more efficient. In the publication, the author concludes that it is beneficial to have the output from multiple analyses in one place. From more than two years of usage, he concluded that the availability of static analysis results contributes significantly to the code review process. However, because

the tool is not publicly available and little information is presented about the way the tool is used, this result is hard to generalize.

During a company-wide event that they called ‘The Google FindBugs Fixit’ [11], Google engineers took a large set of FindBugs generated warnings and assessed them. Over 77% of the warnings generated by FindBugs were identified as real defects and were flagged for fixing. They also observed that, depending on the integration of warnings into the software development process, the usage of static analysis tools may be limited. They concluded that the real value of static analysis tools is in finding problems early and cheaply.

At VMware, Review Bot [14] was created. This tool is a standalone Java application that uses static analysis tools to generate automatic reviews. The tool is triggered when it is assigned as a reviewer in Review Board [21]. It then runs static analysis on the latest dff revision and uploads an automatic review, as if it were a person. The Review Bot can be assigned to a review at any time, to re-run the static analysis. A study of the comments generated by Review Bot showed that a large percentage of comments were accepted as valid.

Panichella *et al.* [50] investigated how warnings detected by static analysis were considered, and to what extent they were removed, during code review. To do this, they mined data from six Java open source projects that use Gerrit [4]. They considered warnings generated by CheckStyle [1] and PMD [6]. Their results indicated that overall, between 6% and 22% of the warnings were removed. However, they also noted that this percentage was significantly higher for certain categories of warnings.

Missing from this related work is the possibility to reproduce the experiment: both SCRUB and Review Bot are proprietary pieces of software that are not available outside of their companies. Furthermore, they do not provide GitHub integration opportunities to their developers. This leaves a large space for an open-source tool which provides a developer with easy GitHub integration; in this thesis, we close this gap by developing an easy-to-use Chrome plugin that fulfills this criteria.

Chapter 3

Approach

The aim of this work is to investigate how static analysis tools can be integrated in code reviews, and how this affects the reviews and the reviewers. In this section, the approach used to answer the research questions is explained. First, we design an experiment and discuss the tested hypotheses and the measured variables. Then, we elaborate on the methods and tools used to collect the data. Finally, the approach to analyzing this data is explained.

3.1 Design of the Experiment

We performed an experiment in which developers used a tool that integrates static analysis results into their code review sessions. The experiment is performed with undergraduate Computer Science students taking the Software Engineering Methods course. Ideally, we would randomly assign students to the treatment and control group respectively. This way, we would obtain two groups that are probabilistically similar to one another before the start of the experiment. However, because the students were graded on their performance during the course project, we felt it would be unfair to have different requirements for each group. Therefore, the students were allowed to self-select the treatment, to minimize the impact of the experiment on the students' performance on the course and the accompanying project. Because the experiment lacks random assignment, it is classified as a quasi-experiment [54]. What this means for the validity of the experiment is explained in more detail in Section 3.1.2.

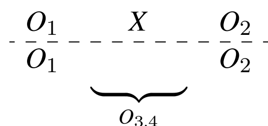

Figure 3.1: The design of the experiment in the notation of Shadish et al. [54]

The design of the experiment is illustrated in Figure 3.1. Each row represents a group of students and the dashed line indicates that students are divided into these groups in a non-random way, in this case through self-selection.

The horizontal dimension represents temporal order from left to right. Every student completed the pretest questionnaire, indicated by O_1. After the pretest questionnaire, all students were given the choice to use a tool that provides additional information during code review. After the treatment period, all students completed the posttest questionnaire, indicated by O_2.

During the treatment period, students were asked to report their attendance to lectures and their access to course materials, indicated by O_3. Finally, comments and commits made in Pull Request sessions on GitHub were collected during the treatment period, indicated by O_4. Students were required to add tags to their comments and commits, categorizing the content of their contributions.

The next section discusses in more detail the organisation of the course in which the experiment is performed. After that, there is a section discussing the validity of the experiment, presenting a list of threats to be considered. Following this, there are sections describing each of the pretest, treatment, posttest, attendance tracking and tag collection (O_1, X, O_2, O_3 and O_4 respectively).

3.1.1 Organisation of the Course

To properly understand the setup of the experiment and the factors contributing or detracting from the validity of its results, it is necessary to understand the context in which the experiment takes place. This section gives an overview of the organisation of the course in which the experiment took place.

The experiment was conducted during the Software Engineering methods course at Delft University of Technology. This course is taught by Dr. Alberto Bacchelli, the supervisor of this thesis, and is part of the second year of the BSc Computer Science curriculum. During this course, students are taught about “the most important software engineering practices needed to build high quality software” [7]. One of these practices is Code Review.

To really learn these practices and to evaluate whether the students have a firm grasp on their application, the students are required to participate in a project. During this project, the students implement a clone of a well-known game in self-selected groups of 4 or 5. Each group can choose to implement one of the following six games.

Space Invaders a 2d arcade game where the player controls a laser cannon and attempts to hold off incoming aliens.

Temple Run a game in which the player controls an explorer who is being chased. The player must collect items and avoid obstacles by jumping, sliding or changing directions.

Bejeweled a tile-matching game in which the player must swap two gems to form groups of three or more adjacent gems of the same type. These gems dissappear and gems fall to take their place. The player can get bonus points by creating bigger groups of gems or causing cascades.

Fishy a game in which the player controls a fish, which can grow bigger by eating smaller fish, but must avoid being eaten by bigger fish.

Bubble Trouble a game in which the player must avoid being hit by bouncing bubbles. The player has to shoot arrows in a vertical line to split the bubbles into halves. Small bubbles dissappear instead of splitting, allowing the player to clear the level.

Bubble Bobble an arcade game in wich the player must shoot bubbles to trap enemies, while jumping along several platforms.

The groups are required to implement their project iteratively. This means that the groups plan the deliverables for each week, and are required to end each iteration with a working product. They are also required to reflect on each iteration and use this reflection as input for their next planning. Following along with the course material and the lectures, each week has a slightly different focus. Weekly grading 'rubrics' are published in advance, so that the students know what is required of them during the iteration. These rubrics can be found in Appendix A.

All source code is required to be versioned on GitHub, in public repositories. Furthermore, development must follow the pull-based development model that was described in Section 2.2. Contributions may only be added to the trunk of the project through GitHub's pull request feature and pull requests may only be merged after they have been reviewed by at least two other team members.

Students are required to develop their games in Java. Every project is required to use Maven [49] as the build tool and its plugins for CheckStyle [1], FindBugs [25] and PMD [6] for static analysis. Finally, each project should be set up for continuous integration using Travis CI.

Besides the project, students are tested on their knowledge of the course material through both a midterm and a final written exam.

3.1.2 Validity of the Experiment

In presenting the design of the experiment, it is important to consider what factors may influence the validity of the measured outcome. By being aware of these threats, an attempt can be made to mitigate or minimize them. More importantly, the threats present can be taken into account when drawing conclusions from the experiment's outcome. Shadish *et al.* [54] divide the threats to validity into *internal* and *external*. The internal validity determines whether the conclusion can be accurately drawn from the experiment; the external validity determines to what extent the conclusion is generalizable. They then identify the following 12 threats to validity.

Threats to internal validity

1. *History:* events other than the treatment occuring between measurements.

2. *Maturation:* processes within the subjects that occur over time.
3. *Testing:* the effect of taking a test on the results of following tests.
4. *Instrumentation:* the influence of (changes in) the measurement tool or method on the measurements.
5. *Statistical regression:* when groups are selected based on extreme pretest scores.
6. *Selection bias:* when the selection of subjects into groups influences the results.
7. *Experimental mortality:* when the number of dropouts across groups is different.
8. *Selection-history interaction, etc:* when the selection into groups creates a differential in the previous factors.

The effects of *history*, *maturation*, *testing*, *instrumentation*, *regression* and *selection* can be mitigated by comparing multiple groups that are affected equally by these factors. Similarly, the threat of *mortality* is minimized when the probability of dropping out is equal between all groups.

The threats of *selection-history interaction*, *selection-maturation interaction*, *selection-testing interaction*, *selection-instrumentation interaction*, *selection-regression interaction* and *selection-mortality interaction* captures the cases where the selection process causes the other factors to not affect all groups equally. In this case, comparison between groups does not cancel out the factors and the effects of these factors can become indistinguishable from the effect of the treatment.

Threats to external validity

9. *Reactive or interaction effect of testing:* the pretest changes the effect of the treatment on the subjects.
10. *Interaction effect of selection bias:* selection may cause the treatment group to be unrepresentative of the general population.
11. *Reactive effects of experimental arrangements:* being aware that they are participating in an experiment changes the effect of the treatment on the subjects.
12. *Multiple-treatment interference:* a subject receiving multiple treatments may not be representative of someone receiving a single treatment because of carry-over.

The threat of the *reactive or interaction effect of testing* can not be mitigated completely. However, we try to minimize this effect by randomizing the order of questions and answers on each survey, and phrase questions so that they are not suggestive of a specific answer. We addressed the *reactive effects of experimental arrangements* by being careful not to frame the tool as an experiment, but rather as something useful for the students during the project. However, the association can not be completely avoided.

The *interaction effect of selection bias* can not be mitigated in this case, because the students were allowed to choose whether or not to use the tool. Instead, we will attempt to measure the differences between the treatment and control group, to get an indication of the extent of this effect. Finally, the effect of *multiple-treatment interference* has to be considered. In the case of this experiment, students only receive a single treatment in all cases. However, we could consider the course material of this and other courses as additional treatments that the students receive, while the general developer population may not have received this or similar treatment. We therefore do not consider generalizing our findings to the whole developer population, but may consider the population of developers who have received similar treatments, i.e. developers who are educated on software engineering principles.

3.1.3 The Pretest (O_1)

The pretest is performed in the form of a questionnaire that all students are required to complete. The pretest is designed for two main reasons: to discover rival hypotheses and to measure the change in students' perceptions over the treatment period.

Because the students self-select the treatment, there is an inherent self-selection bias between the treatment and control group. The pretest contains questions about age, previous attempts at the course, experience with software engineering in general and with specific aspects related to the project. The full list of questions can be found in Appendix B. These factors will help compare the treatment and control group, although it is likely that other factors are present that influence the self-selection.

3.1.4 The Treatment (X)

After taking the pretest, all students were given a short explanation of Octopull. This tool, created especially for this experiment, was designed to measure the effect of having static analysis results available during code review. A technical discussion of the implementation of this tool can be found in Chapter 4.

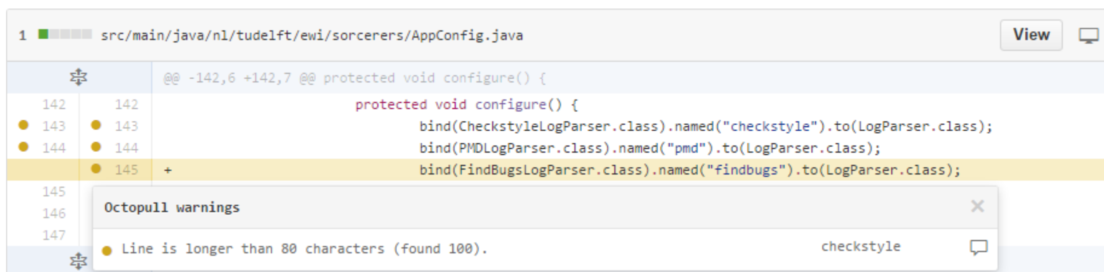

Figure 3.2: An example of the Octopull tool warnings: an orange dot marker indicates warnings for that line, clicking on it will open a full list.

The main function of Octopull is to display the warnings generated from static analysis in the diff view within a pull request on GitHub. For each line in the diff view that

has warnings, an orange dot is placed next to the line number. An example can be seen in Figure 3.2. Note that the left column indicates the 'base' of the diff, while the right column indicates the 'head'. Therefore, warnings in the right column are relevant for the current version of the code, while the left column gives insight in the state before the pull request.

Clicking on the orange indicator provides the user with a list of warnings for that line. Additionally, every warning can be used to generate a review comment with a single click on the button on the right hand side. This generates a review comment on GitHub, visible also to users without the tool. This comment can then be the basis of further discussion about the warning.

3.1.5 The Posttest (O_2)

The posttest, like the pretest, is a questionnaire that all students are required to complete. The posttest has similar questions to the pretest, to determine how students' perceptions have changed over the course of the project. Related to Research Question 1, there are questions to determine the reason students did or did not select the treatment for their project. For members of the treatment group, questions about the user experience and general feedback about the tool are also asked as well. All questions of the posttest can be found in Appendix C.

3.1.6 Attendance and Course Materials (O_3)

Because the selection into groups is not randomized in this experiment, there is a potential threat of *selection-history interaction*. One source of events that can be expected to impact the students' behaviour during code review is the course itself. All students are taking the same course at the same time, so that the course can be expected to influence them equally. However, attendance of the lectures is optional, which may cause individual differences in exposure.

To determine the exposure of the students to the course material, students are asked weekly to indicate whether they attended each lecture, reviewed a video recording of the lecture and/or accessed the course materials available online. With this information, it can be determined whether the treatment and control groups differ significantly in their exposure to the course material. Collection of this data is done through a weekly survey on Google Forms and the students are given access to this survey through BlackBoard, a web application that allows teachers to share course information and announcements with the students.

3.1.7 Comment and Commit Tags (O_4)

To characterize the behaviour of the subjects during code review, all students were required to categorize their comments and commits. Students were able to choose from a list of tags to be added to each comment or commit, that best describe the type of content of that contribution. The list of tags with their description is included in

tag	description
bug	the comment or commit is related to a bug or defect
feature	the comment or commit is related to a new feature
documentation	the comment or commit is related to documentation in the code, most likely a code comment (such as javadoc)
layout	he comment or commit is related to code layout (such as whitespace)
naming	the comment or commit is related to naming of classes, variables, etc.
logic	the comment or commit is related to logical conditions in the code
performance	the comment or commit is related to performance of the code (such as execution speed or memory usage)
organisation	the comment or commit is related to organisation of the code (such as package structure and class responsibilities)
interface	the comment or commit is related to interfaces between parts of the code (such as what methods to call)
user-interface	in contrast to the previous tag, this comment or commit is related to an interface presented to a user (such as a GUI or CLI)
configuration	the comment or commit is related to the configuration of external tools
testing	the comment or commit is related to testing activities
review	this tag only applies to comments and should be used for 'meta-comments' about the review process (please use sparingly)
planning	this tag only applies to comments and should be used for comments related to planning (scheduling, prioritising, etc.)

Table 3.1: The list of tags with their description as provided to the students.

Table 3.1. We created this list based on the categorization of [46], and added some tags that were requested by students, such as 'user-interface' and 'planning'. We also added a tag to capture discussion about the review process itself.

Students were required to provide a list of tags at the end of each commit message or comment, seperated by comma's or whitespace and starting with the '<' sign on a seperate line. This way, these tags can easily be extracted from the text.

By looking at the relative frequency of these tags, it can be determined what type of issues are most commonly addressed during code review. These frequencies can also be compared between groups, as well as over time.

3.2 Hypotheses Formulation

In Section 1.1 we identified the following research questions for this thesis.

> **Research Question 1.** *Are developers willing to use a tool that integrates warnings into their code reviews? If they do, how do they use and experience the tool?*
>
> **Research Question 2.** *What is the effect of displaying warnings on the amount of resolved warnings during code review?*
>
> **Research Question 3.** *What is the effect of displaying warnings on the discussion of code improvements and other concerns during code review?*
>
> **Research Question 4.** *What is the effect of displaying warnings on the developers' perceived priorities during code review?*

Research Question 1 is an open question for which no *a priori* hypothesis is formulated. For the other research questions, we formulate the following hypotheses and determine the corresponding variables to be measured to evaluate them.

Hypothesis 1

- **Null hypothesis ($H1_0$):** There is no significant difference in the amount of resolved warnings per pull request between Octopull users and non-Octopull users.
- **Alternative hypothesis ($H1_A$):** The amount of resolved warnings is significantly different for Octopull users.

Hypothesis 2

- **Null hypothesis ($H2_0$):** There is no significant difference in the distribution of tags between Pull Requests using Octopull and Pull Requests not using Octopull.
- **Alternative hypothesis ($H2_A$):** There is a significant difference in the distribution of tags between Pull Requests using Octopull and Pull Requests not using Octopull.

Hypothesis 3

- **Null hypothesis ($H3_0$):** There is no significant difference in the change of code review priorities from pretest to posttest between Octopull users and non-Octopull users.
- **Alternative hypothesis ($H3_A$):** There is a significant difference in the change of code review priorities from pretest to posttest between Octopull users and non-Octopull users.

Furthermore, to get an indication of the extent of self-selection bias, the following hypothesis is formulated.

Hypothesis X

- **Null hypothesis (Hx_0):** There is no significant difference in the indicated experience on the pretest between Octopull users and non-Octopull users.
- **Alternative hypothesis (Hx_A):** There is a significant difference in the indicated experience between Octopull users and non-Octopull users.

3.3 Measured Variables

Hypothesis	Measured variable
H1	Difference of amount of warnings
H2	Fraction of comments per tag
H3	Difference of priorities from pre- to posttest
Hx	Reported experience and proficiency

Table 3.2: The corresponding variables for each hypothesis.

Table 3.2 lists the measured variable for each hypothesis. For each of the hypotheses, we will measure the corresponding variable in the treatment and control group. These measurements yield two datasets

$$x_1, x_2, \ldots, x_n \qquad \text{and} \qquad y_1, y_2, \ldots, y_m$$

for the treatment and control group respectively, which are the realization of independent random samples

$$X_1, X_2, \ldots, X_n \qquad \text{and} \qquad Y_1, Y_2, \ldots, Y_m$$

from two distributions. Each hypothesis can then be reformulated in terms of these random distributions. In the following sections, each variable is discussed in turn.

3.3.1 Difference of amount of warnings

The relevant variable for *Hypothesis 1* is the difference of warning density per pull request. This variable is measured for each merged pull request as follows.

Let w_c be the number of lines and number of warnings visible in the diff of the pull request for commit c, where $c \in \{base, head\}$. Then, the measured variable $\Delta = w_{base} - w_{head}$. Additionally, we consider the fractional difference $r = w_{base}/w_{head}$.

If these values of Δ or r are seperated for projects that have Octopull enabled versus projects that do not, two datasets are obtained: $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$, for the control- and treatment group respectively, which are the realization of independent

random samples from two distributions, with unknown mean and variance. Hypothesis 1 can then be formulated in terms of the means of these distributions.

$$H1_0 : \mu_t = \mu_c \qquad \text{and} \qquad H1_A : \mu_t \neq \mu_c$$

where μ_t is the mean of the distribution for the treatment group and μ_c is the mean of the distribution for the control group.

3.3.2 Fraction of comments per tag

The relevant variable for *Hypothesis 2* is the fraction of comments for each tag. The variable is measured for each tag per pull request as follows.

For each pull request, let $N = \sum_{l \in T} c_l$ be the total amount of tags, where T is the set of all allowed tags, and c_l is the amount of tags for tag l. Then, the measured variable is $p_l = \frac{c_l}{N}$ for each $l \in T$.

If these values of p_l are seperated for pull requests of the treatment group versus those of the control group, two datasets are obtained for each tag l: $x_{l,1}, x_{l,2}, \ldots, x_{l,n}$ and $y_{l,1}, y_{l,2}, \ldots, y_{l,m}$, for the control- and treatment group respectively, which are the realization of independent random samples from two distributions, with unknown mean and variance. Hypothesis 2 can then be formulated in terms of the means of these distributions.

$$H2_0 : \mu_{l,t} = \mu_{l,c} \qquad \text{and} \qquad H2_A : \mu_{l,t} \neq \mu_{l,c} \text{ for } l \in T$$

where $\mu_{l,t}$ is the mean of the distribution for tag l in the treatment group and $\mu_{l,c}$ is the mean of the distribution for tag l in the control group, and T is the list of tags.

3.3.3 Difference of code-review priorities

The relevant variable for *Hypothesis 3* is the difference of perceived code-review priorities. That is, the difference between the ranking of priorities in code review from pre- to posttest.

Let $a_{t,1}, a_{t,2}, \ldots, a_{t,n}$ be vectors of the ranks each option was assigned in the pretest by the students of the treatment group and let $a_{c,1}, a_{c,2}, \ldots, a_{c,m}$ be the ranks assigned by the control group. Similarly, let $b_{t,1}, \ldots, b_{t,n}$ and $b_{c,1}, \ldots, b_{c,m}$ be the score of the same item on the posttest. Then, the measured variables are $x_i = \rho_i$ where ρ_i is the Spearman correlation [28] between a_i and b_i. These values are the realization of independent random samples from two distributions, with unknown mean and variance. Hypothesis 3 can then be formulated in terms of the means of these distributions.

$$H3_0 : \mu_t = \mu_c \qquad \text{and} \qquad H3_A : \mu_t \neq \mu_c$$

where μ_t is the mean of the distribution for the treatment group and μ_c is the mean of the distribution for the control group.

3.3.4 Reported experience and proficiency

The relevant variable for *Hypothesis X* is the reported experience and proficiency of the students from the pre-test. For each of the subjects, students rated their experience on a 5-point Likert scale [45]. For each question, we count the number of students that chose each answer. If split by treatment and control group, this yields a contingency table as seen in Table 3.3. The hypothesis can then be formulated as follows.

$$Hx_0 : \forall r \in \{1,2,3,4,5\}\ P(rating = r \mid \text{in treatment group}) = P(rating = r \mid \text{in control group})$$

$$Hx_A : \exists r \in \{1,2,3,4,5\}\ P(rating = r \mid \text{in treatment group}) \neq P(rating = r \mid \text{in control group})$$

	1	**2**	**3**	**4**	**5**	**total**
treatment	x_1	x_2	x_3	x_4	x_5	n
control	y_1	y_2	y_3	y_4	y_5	m
total	x_1+y_1	x_2+y_2	x_3+y_3	x_4+y_4	x_5+y_5	$n+m$

Table 3.3: An abstract contingency table of both groups versus the occurences of each rating.

3.4 Data Collection

The collected data comes from several sources. In the next subsections we discuss the multiple surveys, the collection of tagged comments and warnings respectively.

3.4.1 Surveys

The pretest survey was created using SurveyGizmo [8]. This web-application allows all results to be downloaded in comma seperated values format, which is suitable for analysis in several tools such as Microsoft Excel and the statistical software R. The posttest survey was created in Google Forms, which allows downloading all entries in that format as well. Attendance data was also collected based on a weekly survey. The students received a link to a Google Forms page each week, where they could indicate their attendance for that weeks lectures.

3.4.2 Tag Collection

To collect data regarding comment and commit tags, a custom tool was implemented. The details about the implementation can be found in Chapter 4. A list of the comment and commit tags was given in Section 3.1.7. Students are required to provide a list of these tags, preceded by a ‘<’ sign in each of their comments and commits. Because the tags adhere to a standard format, they can be accurately extracted from comment and

commit messages. To do so, the tool accesses GitHub to retrieve the comments and commits for each pull request. The tags are then extracted and stored, grouped by pull request.

3.4.3 Warning Collection

The warning collection consists of two parts: the Octopull tool that is implemented for the treatment group, and the warning collector for other projects. The implementation of each tool is explained in Chapter 4. For projects in the treatment group, warning collection is quite straight-forward. Because Octopull requires access to a list of warnings for each relevant commit, these warnings are available directly from the database of Octopull. For the projects of the control group, the process is slightly more complex. However, because all projects are required to use Maven, a standard solution is still possible: a script is created to retrieve the relevant commits from GitHub for both groups, and execute the Maven plugins for FindBugs, PMD and CheckStyle. Each plugin outputs its results as an XML file, ready for analysis.

After the data has been collected, each of the hypotheses can be evaluated. The appropriate statistical methods are applied using the statistical software R [27] as follows.

3.5 Data Analysis

To test if two groups of random samples come from the same distribution, for random samples that are not randomly distributed, we apply the Mann-Whitney-Wilcoxon test [38]:

```
# import the coin package
> library(coin)
# perform the test
> test = wilcox_test(variable ~ group, data=set)
> pvalue(test)
```

We obtain its Cohen's *r* effect size as follows

```
> r = statistic(test) / sqrt(length(set$variable))
```

When testing multiple hypotheses in order to draw a single conclusion (e.g. when we reject our null hypothesis when any of the null hypotheses are rejected), we increase the probability of committing a Type I error. In order to adjust this, we can apply the Benjamini-Hochberg procedure to the p-values of all tests [18] as follows.

```
> p.adjust(p_values, 'BH')
```

When testing the correlation between two nominal variables, we can employ the Chi-square test [28]. We first create a contingency table of two variables, then test them [28] as follows.

```
> tbl = table(set$var1, set$var2)
> chisq.test(tbl)
```

Chapter 4

Implementation

To perform the experiment from the previous chapter, a tool that integrates static analysis and code review was needed for the treatment group. Because no such tool currently exists for GitHub, and no open-source implementation was available that could be adapted, we implemented this tool ourselves. Our open-source implementation, called Octopull [1], provides a clear view of static analysis violations directly in GitHub's pull requests. The tool integrates seamlessly with the existing user-interface and requires little configuration to work.

In this chapter, the implementation of this tool is explained. First, the overall architecture of the solution is explained. Then, each component is described seperately, going more into the details of their implementation and/or configuration.

4.1 Architecture

In Figure 4.1, an overview is given of the implemented architecture. This high-level overview contains the main components and interactions of the solution.

Because the solution is meant to integrate into the code-review process on GitHub, the interaction between the developers and GitHub was modeled first. Starting at the bottom left of the diagram, developers frequently push their changes to GitHub (). Code review is then started by creating a Pull Request (). The developers then interact with this Pull Request through the GitHub user-interface by viewing the changes, placing comments and pushing new changes, until the Pull Request is accepted or rejected.

To integrate the Static Analysis results into the code review session, we decided to display static analysis warnings in the difference view of the Pull Request user-interface (). Initially, static analysis results were displayed as review comments (), i.e. comments on GitHub related to specific lines of the diff. However, we soon found that this became cumbersome for larger amount of warnings. Because GitHub provides no other way to annotate the diff, we implemented a plugin for the Chrome browser () to modify GitHub's user-interface on the client-side ().

[1] https://github.com/rmhartog/octopull/

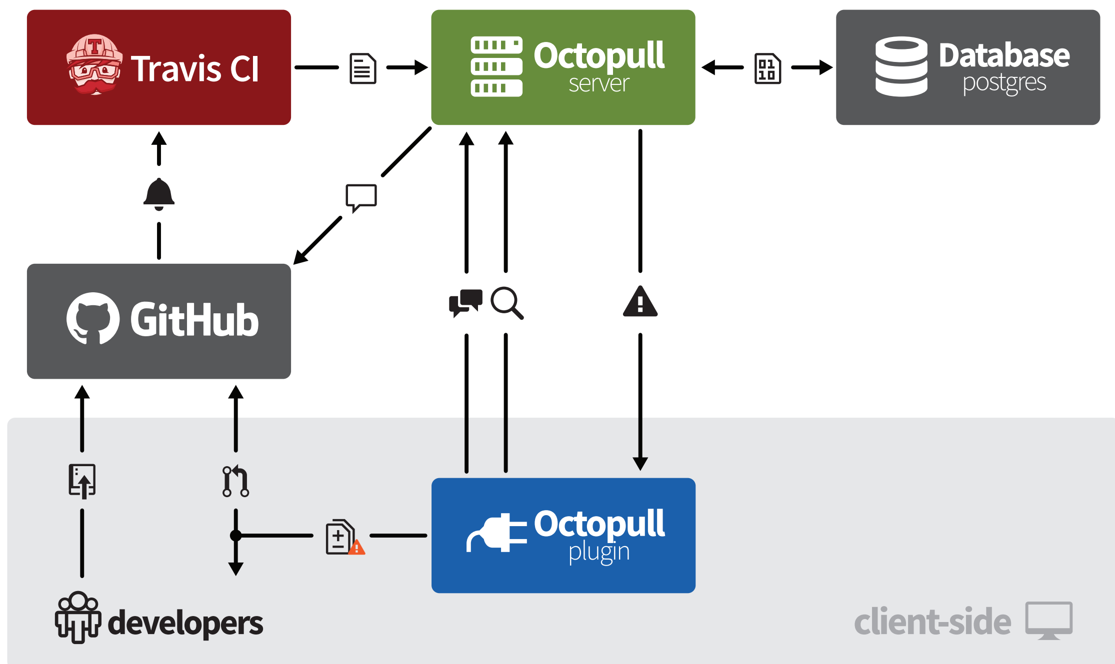

Figure 4.1: A high-level overview of the implemented solution.

To avoid the complexity of creating infrastructure that fetches the changes for each Pull Request and performs static analysis, we decided to leverage Travis CI, because it is commonly used with GitHub, straight-forward to configure and free for open-source projects. When new changes are pushed to the GitHub repository, a notification is sent to Travis (🔔). Travis then retrieves the changes and performs static analysis, using the configured tools. It then notifies the Octopull server, which obtains the log output for each job (📄), parses the static analysis results and stores this data (▯) in a database (🛢).

Whenever the user browses GitHub, the plugin communicates the current page to the server (🔍). If the page corresponds to a Pull Request, the server loads a list of warnings from the database and sends them (⚠) to the plugin to be displayed. The user can then review the annotated diff for warnings (▯). Finally, if a user wants to further discuss a warning, he can choose to turn the warning into a review comment. The plugin then sends a request (💬) to the server, which creates the comment (🗨) on the user's behalf.

In the next sections, the implementation and/or configuration of each of the components is described in more detail.

4.2 Travis configuration

For the Octopull server to receive the static analysis results correctly, some configuration of Travis is required. To enable Travis, the owner of the repository has to register with Travis using his/her GitHub credentials. Then, the Travis website displays a list of the user's projects and allows them to be enabled or disabled. An example can be seen in Figure 4.2. After the project has been enabled, a configuration file called `.travis.yml` must be pushed to the repository. This file describes the configuration of the project in YAML format [17]. An example of this configuration file can be found in Figure 4.3. The explanation of each of the settings is explained below.

Figure 4.2: Travis CI switches for each of the user's projects.
Note: this image was edited for space.

```
1  language: java
2  notifications:
3    webhooks:
4      - http://octopull.rmhartog.me/api/travis/webhook
5  before_install:
6    - '[ "$BUILD_PR_BRANCH" = "true" ] && { [ "$TRAVIS_PULL_REQUEST" != "false" ] || exit 1; } && git checkout HEAD^2↩
         && echo "OCTOPULL_SHA=$(git rev-parse HEAD)"; true'
7  env:
8    - BUILD_PR_BRANCH=true
9    - BUILD_PR_BRANCH=false
10 matrix:
11   allow_failures:
12     - env: BUILD_PR_BRANCH=true
13 after_script:
14   - echo "== CHECKSTYLE_RESULT =="; cat "target/checkstyle-result.xml"; echo "== END_CHECKSTYLE_RESULT =="
15   - echo "== PMD_RESULT =="; cat "target/pmd.xml"; echo "== END_PMD_RESULT =="
16   - echo "== FINDBUGS_RESULT =="; cat "target/findbugsXml.xml"; echo "== END_FINDBUGS_RESULT =="
```

Figure 4.3: An example Travis configuration file for a project using Octopull.

Because Octopull currently only supports Java projects using Maven, the `language` setting is set to java on line 1. Furthermore, to notify the Octopull server about each build, the `notifications` section (line 2-4) contains an entry for a webhook pointed to a special webhook endpoint of the Octopull server. The `after_script` entries (line 13-16) copy the contents of the Checkstyle, PMD and FindBugs reports to the log file, delimited by tokens to facilitate detection of these contents.

To understand the significance of lines 5-12, some knowledge about the working of Travis is required. By default, Travis creates builds in two situations: when commits are pushed to the repository or when a Pull Request is updated. In the case of a push, Travis builds the commits that were pushed to the repository. In the case of a Pull Request, Travis builds the merge commit between the Pull Request's head and its base.

The underlying Git structure of a Pull Request is illustrated in Figure 4.4. The diff that is displayed in a Pull Request is the difference between the head and the merge-base (respectively labeled E and A in the figure). Therefore, to annotate this diff, static analysis results for both of these revisions are needed. In the case of a single repository, this is not a problem, as these commits are built by Travis when they are pushed to the repository. However, when the Pull Request is made from a different repository, such as when working with forks, commits D and E will not be pushed to the repository and thus not built by Travis.

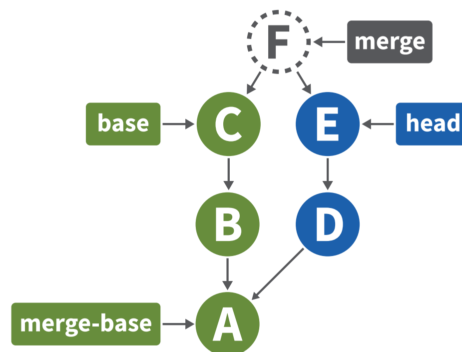

Figure 4.4: The underlying Git structure of a Pull Request.

To overcome this, we configure Travis to build the head commit in addition to the merge commit for every Pull Request. This is acheived by leveraging Travis' build matrix. Every Travis build is made up of one or more jobs. Each of these jobs is run independently and together they determine the build result. Line 7-9 of the configuration file add a dimension to the build matrix: an environment variable named `BUILD_PR_BRANCH` with the possible values `true` and `false`. This doubles the number of jobs in the matrix. Line 5-6 then define the `before_install` step, which is the first step of every job. This step is a chain of the following commands:

`[ "$BUILD_PR_BRANCH" = "true" ]`	test if the `BUILD_PR_BRANCH` variable is set to `true`.
`&& { [ "$TRAVIS_PULL_REQUEST" != "false" ] || exit 1; }`	If it is, test if we are building a Pull Request and terminate the step with a failure if we're not.
`&& git checkout HEAD^2`	If we are, check out the second parent of the current commit.
`&& echo "OCTOPULL_SHA=$(git rev-parse HEAD)"`	If it is checked out, retrieve the SHA of the current HEAD and echo it to the log.
`; true`	End the script with a success code (unless exit was called).

For a Pull Request build, the job continues after the step, but it will work on the head commit instead of the merge commit if `BUILD_PR_BRANCH` equals `true`. Because Travis is unaware that the job works on a different commit, the SHA is echoed into the log file for later parsing. For a push build, the job terminates with a failure if `BUILD_PR_BRANCH` equals `true`. Line 10-12 of the configuration indicate that this should be considered an 'allowed failure', so that the whole build is not marked as failed.

This configuration ensures the correct commits are built, that static analysis results are stored in the log file and that the Octopull server is notified. The Octopull server can then retrieve the log and parse the relevant information from it.

4.3 Octopull Server

The Octopull server was introduced to receive, process and store static analysis results, and distribute them to clients when required. Additionally, its central role makes it suitable for gathering usage data. In this section, the implementation of the Octopull server is explained in more detail.

Octopull server is implemented as a stand-alone Java application. Its architecture is inspired by the Hexagonal Architecture [22]. At the core of the application is a small domain model [30] that represents the relevant entities. Operations on the domain model are exposed as Use Case objects with a single `execute` method. External services, such as GitHub and Travis, are represented as interfaces in the domain model, and implemented in a separate package. Similarly, the underlying storage is represented in the domain model with interfaces following the Repository pattern [30]. These interfaces are also implemented in a separate package.

A high-level overview of the architecture is shown in Figure 4.5. The domain model is central to the application, and has no outgoing dependencies. A small web application is built on top of the Jersey framework [5] and runs inside an embedded Jetty [34] webserver. This layer exposes the use cases over the HTTP protocol. A package provides the ability to retrieve logs from Travis. Similarly a package provides the necessary services to communicate with GitHub. This package is a wrapper around the egit-github library [3]. Finally, the repository interfaces are implemented in a package using the Java Persistence API (JPA) [41]. The Hibernate library [15] is included to provide JPA functionality.

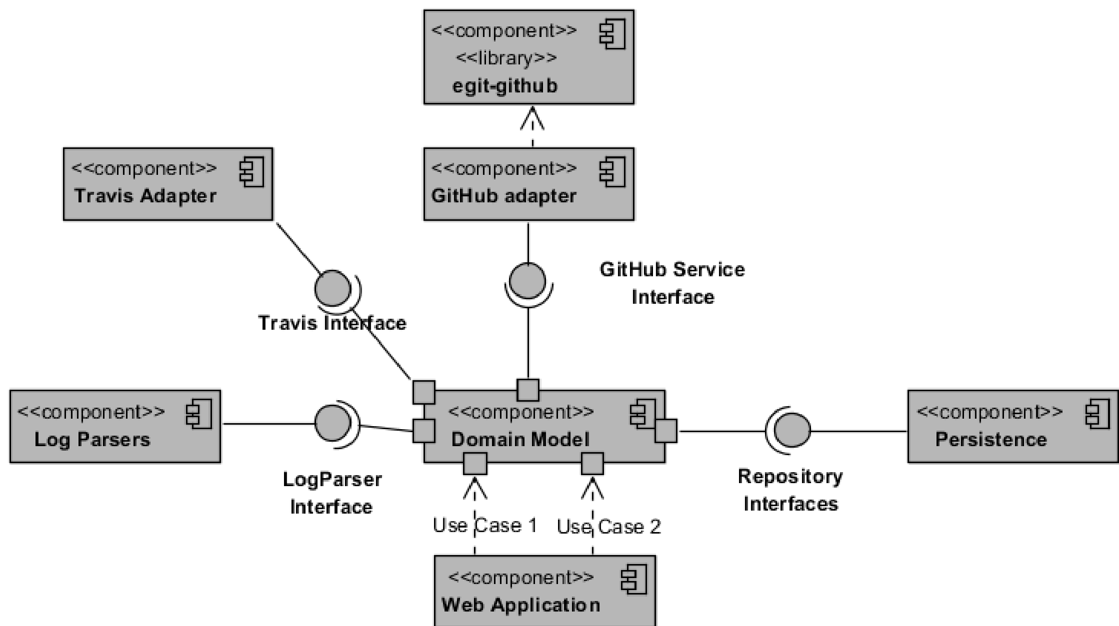

Figure 4.5: A high-level overview of the architecture of the server project.

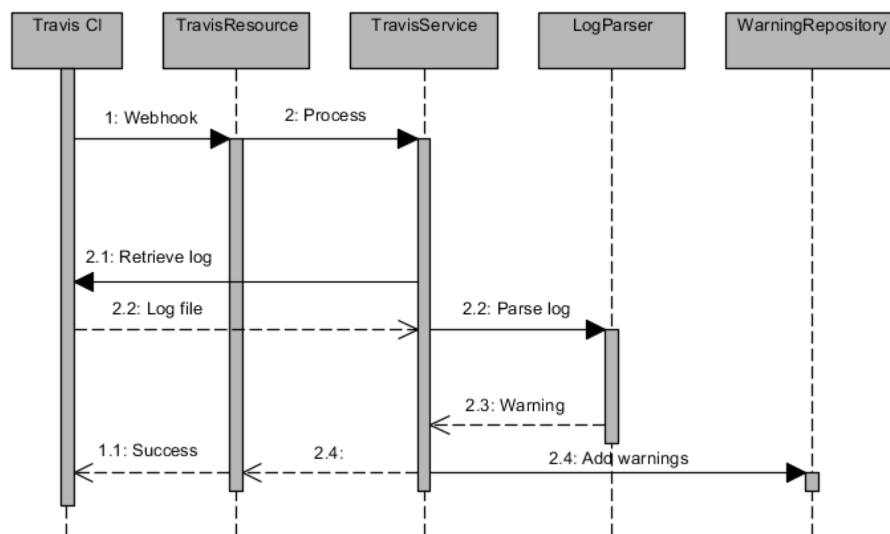

Figure 4.6: An overview of the interactions that occur when a Travis build completes.

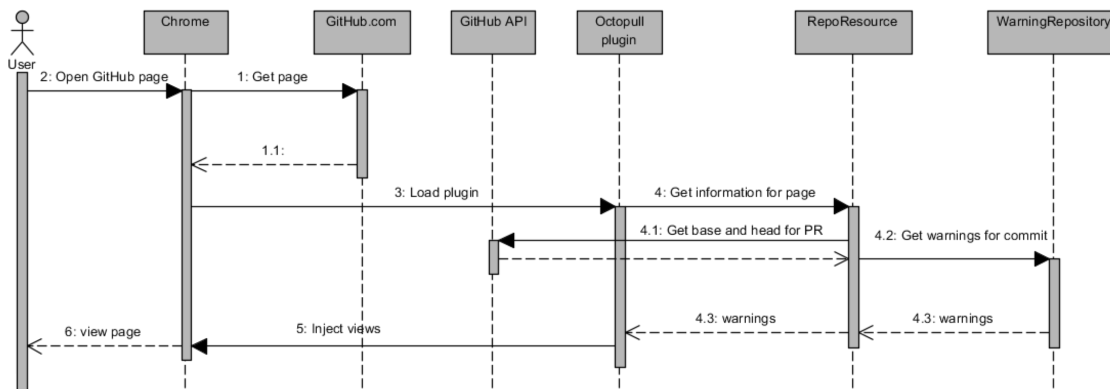

Figure 4.7: An overview of the interactions that occur when an Octopull users browses GitHub.

In Figure 4.6 the interaction between the different components is shown when a webhook from Travis is received.

1. A webhook notification from Travis is received.
2. It is routed to the `TravisResource`, which parses the request and sends it to the `TravisService`.

2.1. The `TravisService` is retrieves the log file from Travis.

2.2. All warnings are parsed from the job log.

2.4. The warnings are added to the `WarningRepository`.

-. Hibernate adds the warnings to the database.

In Figure 4.7 the interaction between the different components is shown when the user browses GitHub using the plugin.

1. The user opens a GitHub page.

3. The plugin is loaded.

4. The plugin contacts the Octopull server's `RepoResource` to retrieve the relevant information for the current page.

4.1. The Octopull server retrieves information about the current pull request from the GitHub API.

4.2. Based on this information, the relevant warnings are retrieved from the `WarningRepository`.

4.3. The list of warnings is returned to the plugin.

5. The plugin injects additional markup into the page.

6. The user is presented with the warnings.

Environment variable	Description
POSTGRES_URL	The url of the postgres database.
GITHUB_TOKEN	A GitHub token used to authenticate with the Travis API.
GITHUB_CLIENT_ID	The Client ID of the application registered with GitHub.
GITHUB_CLIENT_SECRET	The Client secret of the application reigstered with GitHub.

Table 4.1: The environment variables used to configure the Octopull Server.

4.3.1 Configuration and deployment

The Octopull Server requires a postgres database to be deployed. Furthermore, it requires a token for a GitHub user to authenticate with Travis, and the client identifier and secret for the application registered with GitHub.

The Octopull Server project uses Maven as a build tool, which can be used to create a packaged jar-file with dependencies. When this jar-file is run, it will provide the web-service on port 8080. The configuration is read from the environment variables in Table 4.1. During the experiment, the Octopull Server was hosted on the DigitalOcean cloud platform. An NGINX proxy was used to route all http traffic addressed to `https://octopull.rmhartog.me/api/` to port 8080 internally.

4.4 Octopull Plugin

During the initial development, the Octopull server annotated Pull Requests directly by placing a review comment for each warning. However, this quickly became cumbersome when large amounts of warnings were detected. To provide a more user-friendly view of the warnings, a plugin was developed to extend the GitHub user-interface.

We chose to implement a plugin for Google Chrome, as it has the largest market-share [9] and a good plugin mechanism. This mechanism allows so-called 'extensions' to inject additional markup and scripts into each page for which the URL matches a certain pattern. The Octopull plugin is written in JavaScript and injected in every page belonging to the `github.com` domain.

The plugin is divided into the following components:

- *Agent*: which encapsulates the requests to the Octopull server.
- *Warning-Model*: which encapsulates the warnings to be displayed.
- *View*: which encapsulates multiple visible child-views and listens for page updates.
- *Message-View*: which displays error messages from the server.
- *Diff-View*: which injects the warnings into the pull request diff.

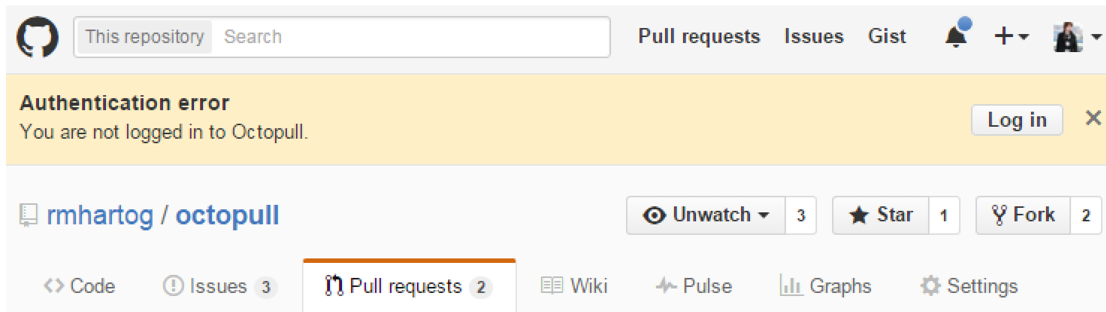

Figure 4.8: The Octopull plugin prompts the user to log in to the server.

- *Overview-View*: which displays an overview of the warnings and allows them to be filtered.

Each component is written as a CommonJS module [26], which is a module system for JavaScript that allows an application to be broken down into modules to improve the maintainability, by providing encapsulation and separation of concerns. Because browsers do not yet support loading such modules directly, Browserify [10] is used to convert these seperate modules into a single script suitable for injection into the browsers' pages.

Whenever the user opens a new page within the `github.com` domain, this script is injected. Execution is then started from scratch. However, a lot of navigation on GitHub only partially updates the page. To do this, they leverage ajax requests [52] and update the current url using pushState [52]. In this case, the script is not reinjected, and the *View* module must detect that a navigation event happened. To do so, the *View* module listens for changes to specific DOM elements, which are used by GitHub to indicate that the page is partially updating. After the update is complete, the script is reattached to the updated page.

The script adds three main components to the GitHub user-interface. First of all, a status indicator and error message window is added on top of the page, so that users can be notified of events in the plugin. An example of this view is shown in Figure 4.8. In this example, the user is prompted to confirm their identity with the Octopull Server.

Second, when the user is logged in and the user navigates to the 'files view' of a Pull Request, the Octopull plugin displays a collapsible overview of the warnings for that Pull Request, in the bottom right corner of the page. An example of this view is shown in Figure 4.9. In this overview, the user can browse all the warnings and jump to a particular instance on the page. They can also filter the warnings that are displayed to narrow down the relevant issues quickly.

Finally, the Octopull plugin displays the warnings in each 'diff view' of the Pull Request, as can be seen in Figure 4.10. The orange dot indicates one or more warnings on that line. These markers are placed in either the left or right column of line numbers, indicating that they belong to the base and head version of the code respectively. Clicking on this marker will display a list of warnings with their description and the tool that

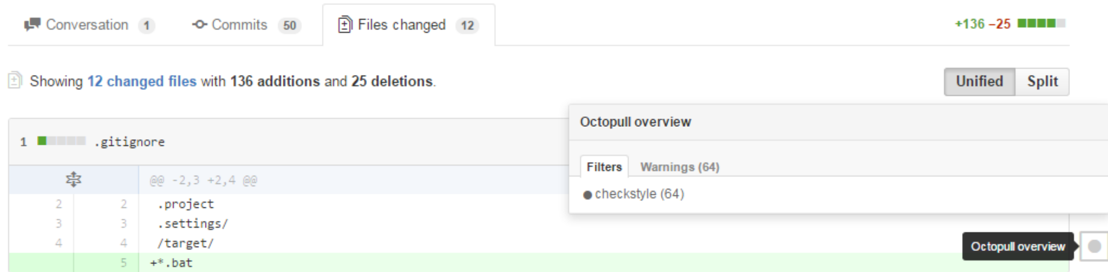

Figure 4.9: The Octopull plugin provides a filterable overview of the warnings for this Pull Request.

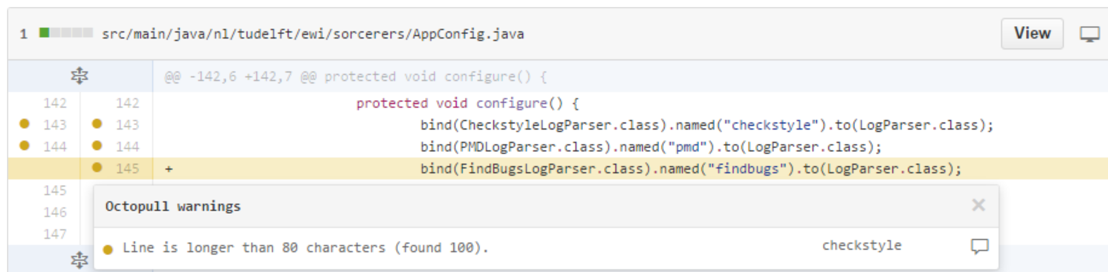

Figure 4.10: The Octopull plugin displays the warnings in the 'diff view'.

generated them. If the warning is still present in the latest version of the code (i.e. the head version), the popup also provides a button that adds an auto-generated comment on the same line of the diff. Figure 4.11 shows an example of such a comment, which is visible in both the 'conversation view' and the 'files view', even for users not using Octopull.

4.5 Data retrieval

GitHub provides an API, through which data about repositories, pull requests, comments and commits can be retrieved. Because all the projects are hosted in public repositories, data about commits, pull requests and comments can be downloaded without additional requirements. After retrieving a list of all pull requests, we retrieve a list of commits for each pull request, and recursively retrieve the list of 'commit statuses' for each commit. These commit statuses are then used to determine which commits were built and validated by Travis CI. Because Octopull users can only see violations on commits that are built by Travis, we select the first and last built commits for each pull request. Because not all projects are properly configured for Octopull, we download all these commits and modify their Travis configuration to output FindBugs, CheckStyle and PMD violations. We push these commits to seperate branches on a new repository that is enabled with Travis. This triggers Travis to build all these commits, as if all projects had Octopull enabled. Finally, we retrieve the log output for each build, as well as the

rmhartog commented on the diff just now

src/main/java/nl/tudelft/ewi/sorcerers/AppConfig.java View full changes

```
@@ -116,6 +123,9 @@ protected void configure() {
116 123                 bindFactory(GitHubClientFactory.class).to(GitHub
117 124                 bindFactory(PullRequestServiceFactory.class).to(
118 125                 bindFactory(CommitServiceFactory.class).to(Comm
    126 +
```

rmhartog added a note just now Owner

Auto-generated comment

```
Line has trailing spaces.
```

```
@@ -116,6 +123,9 @@ protected void configure() {
116 123                 bindFactory(GitHubClientFactory.class).to(GitHubClient.class).in(RequestScoped.class);
117 124                 bindFactory(PullRequestServiceFactory.class).to(PullRequestService.class).in(RequestScoped.class)
118 125                 bindFactory(CommitServiceFactory.class).to(CommitService.class).in(RequestScoped.class);
    126 +
```

rmhartog added a note 36 seconds ago Owner

Auto-generated comment

```
Line has trailing spaces.
```

Add a line note

Figure 4.11: A comment generated from an Octopull warning in the 'conversation view' (*top*) and in the 'files view' (*bottom*).

'diff' between each of these commits and the pull request base.

Chapter 5

Results

In this thesis we sought to answer the question whether the integration of Static Analysis with Code Review increases the effectiveness and/or efficiency of the Code Review process. In Section 3.1 we have the design of our experiment. In Chapter 4 we have described the implementation of the Octopull tool used in this experiment. We then performed the experiment with a group of students and collected a dataset. In this section we analyze this dataset.

We will give an exploratory overview of the dataset that was collected, before we evaluate the formulated hypotheses in the context of this dataset.

5.1 Overview of the collected data

We have collected several types of data: (1) students completed the pre- and posttest questionnaires. (2) We mined all pull requests from the student repositories, and their corresponding comments and commits, from which we extracted the corresponding tags. Furthermore, (3) we asked students to record their attendance in online forms. Some key figures of the dataset are shown in Table 5.1. Because almost no students completed all the attendance forms, we were not able to use them for our analysis.

Metric	Value
Number of students that completed pretest	160
Number of students that completed posttest	68
Number of students that installed Octopull	63
Number of Pull Requests collected	2748
Total number of tags used	14812

Table 5.1: Key size figures about the collected dataset.

5.2 Developer interest and usage

RQ1: Are developers willing to use a tool that integrates warnings into their code reviews? If they do, how do they use and experience the tool?

To determine whether developers are interested in using the tool, we look at the percentage of students that chose to use the tool. We retrieved the list of all GitHub usernames that logged into the Octopull server and cross-referenced them with the GitHub usernames from the pretest. Out of the 160 students that filled out the pretest, 63 (39%) installed Octopull and logged in at least once.

Out of the 37 students that indicated to have used Octopull on the posttest, 10 indicated they would like to use it again, and 18 indicated they would not. The other 9 students did not answer this question. The students that would use the tool again all indicated similar reasons, such as "Octopull diplays warnings in the pull request, so we don't have to do that ourselves." and "To see static analysis results in github seemed like a neat thing to have.". Other students indicated that they were required to use Octopull, even though this was not the case. The students that did not use the tool indicated that they did not know what the tool was for, or that they thought it would be too much effort to set it up. One student wrote "Actually I haven't even looked at octopull because we were swarmed with useless tools like checkstyle, pmd, findbugs which most of the time only made our code more ugly (especially checkstyle), so I decided to not waste my time with any more tools provided by the lecturer (except for inCode, I like that one)."

Besides displaying the violations within the GitHub interface, the Octopull plugin provides users with the ability to convert any violation that is visible into a review comment. Users with and without Octopull can then discuss this comment further and decide on the appropriate course of action. This feature was only used in 3 projects, for a total of 40 times.

Takeaway It seems from these observations that the students were not very willing to try a new tool (only 39%). Some students were more positive, however, and saw benefit in using a tool like Octopull.

5.2.1 The effect on resolved warnings

RQ2: What is the effect of displaying warnings on the amount of resolved warnings during code review?

To determine if there is a relation between displayed warnings and the resolution of these warnings, we collected the set of violations from static analysis tools for the beginning and end of each code review session (i.e. every pull request). These sets are filtered to only contain the violations that fall within the visible lines on the complete diff of the pull request. In that way, we obtain the list of violations a user would see if

they were using Octopull. Subtracting the end number from the start number yields the absolute difference per pull request.

In Figure 5.1, a boxplot of these values is shown for each of the static analysis tools. It is clear that there is a difference between Octopull and non-Octopull projects. We see that there are a lot of cases where the number of violations has actually increased over the lifetime of the pull request.

For FindBugs, very few violations were detected overall, as can be seen from the figure. For PMD, we see that pull requests without Octopull have a larger variability in the number of violations added or removed. Most notably however, we see that pull requests using Octopull tend to introduce more violations, or remove less. We would expect that making violations more visible would prompt users to remove more, and so this last finding seems counter-intuitive. We investigate it further by looking at the fraction $\Delta_v = v_{end}/v_{start}$ of remaining violations instead. In Figure 5.2, we see what we would expect for PMD and FindBugs: their shape is similar to that of the absolute difference. For CheckStyle, however, we see an interesting difference: whereas the absolute difference tends to be larger for pull requests using Octopull, the fractional difference tends to be smaller. This means the average number of CheckStyle violations is larger for pull requests using Octopull. Looking at the mean start and end values shows us that this is indeed the case: the mean start and end value for CheckStyle are 47 and 64 violations respectively for pull requests not using Octopull, and both are 92 violations for pull requests that use Octopull.

Because we can not assume the data to be normally distributed, we apply the Mann-Whitney-Wilcoxon test [38] to the collected data. We find that the absolute difference of CheckStyle violations differs significantly between Octopull and non-Octopull pull requests at a significance level of 5% ($p = 0.025$). The absolute differences for other tools are not statistically significant. The fractional difference is not found to be statistically significant for any of the tools.

This significant difference for the Checkstyle tool leads us to reject hypothesis $H1_0$: *There is no significant difference in the amount of resolved warnings per pull request between Octopull users and non-Octopull users.* in favor of the alternative $H1_A$: *The amount of resolved warnings is significantly different for Octopull users.*

Takeaway Even though we reject the null hypothesis, we are not convinced of the effect of Octopull on the amount of resolved warnings. Because the results are only significant for one tool, and the fractional difference does not show significance, we suspect that this effect can be attributed to an external factor, such as problems with the CheckStyle configuration.

5.2.2 The effect on discussion

RQ3: What is the effect of displaying warnings on the discussion of code improvements and other concerns during code review?

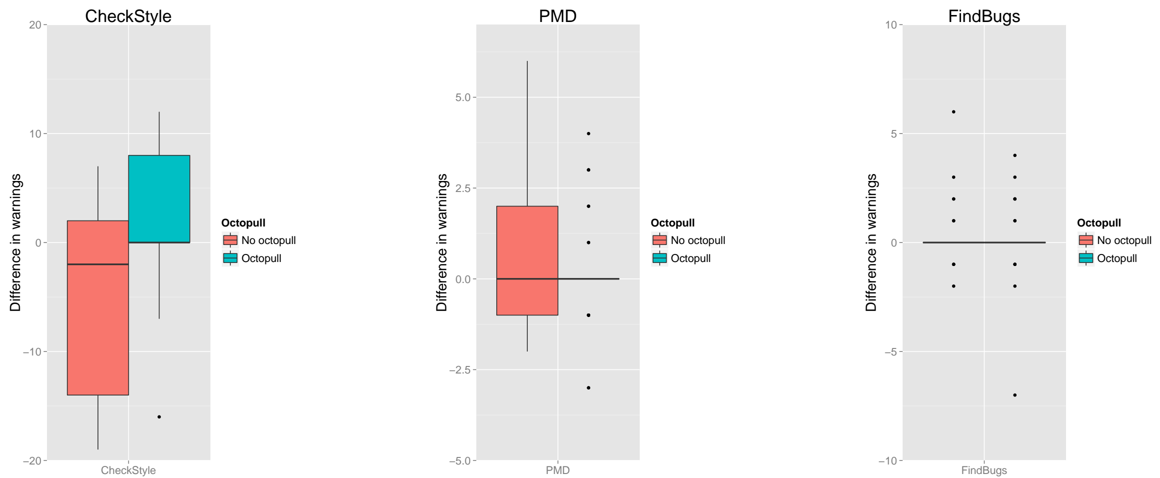

Figure 5.1: Boxplots of the absolute difference in violations before and after the pull request.

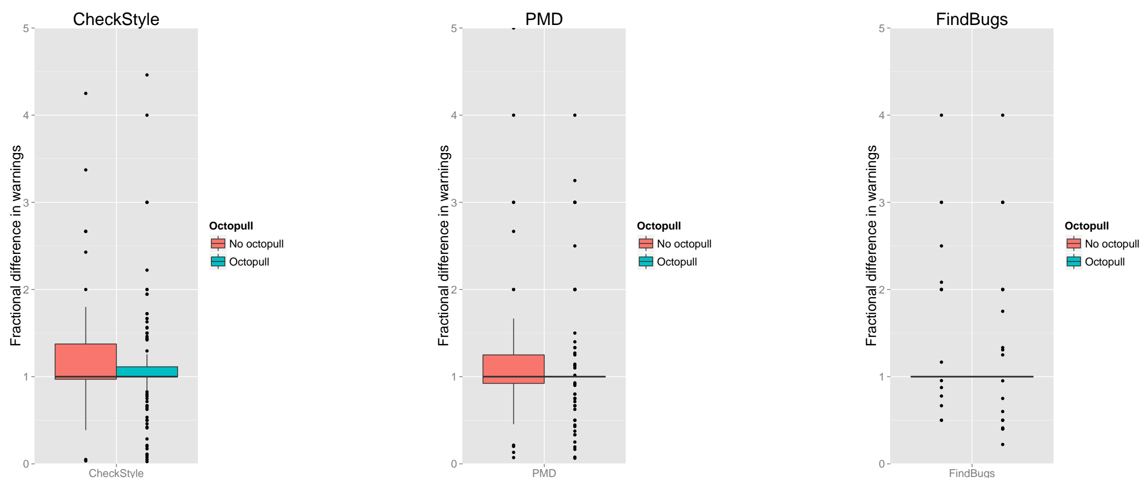

Figure 5.2: Boxplots of the fractional difference in violations before and after the pull request.

To evaluate the relation between the usage of Octopull and the discussion on code reviews, we required students to categorize their comments and commits based on their content, as was explained in section 3.1.7. We then analyze the fraction of comments on each pull request that falls within a certain category to determine if they differ significantly on pull requests with or without Octopull.

We gathered the number of occurences for each tag per pull request and divided them by the total number of tags for that pull request to obtain the relative distributions. We then split these into two groups, for pull requests using Octopull and pull requests

tag	p-value	r-value	adjusted value
bug	2.522E-02	0.0544	3.210E-02
feature	1.089E-03	0.0794	1.906E-03
documentation	4.176E-09	-0.1428	2.180E-08
layout	2.940E-05	-0.1015	6.859E-05
naming	1.190E-06	-0.1180	4.166E-06
logic	2.294E-04	-0.0895	4.587E-04
performance	2.159E-05	-0.1032	6.044E-05
organisation	4.838E-03	0.0685	7.526E-03
interface	4.670E-09	-0.1423	2.180E-08
user-interface	3.177E-01	-0.0243	3.422E-01
configuration	3.010E-02	-0.0527	3.511E-02
testing	7.815E-03	-0.0646	1.094E-02
review	7.377E-22	-0.2334	1.033E-20
planning	7.448E-01	-0.0079	7.448E-01

Table 5.2: The p-value and r effect size for a Mann-Whitney-Wilcoxon-test between Octopull and non-Octopull usage numbers for each tag, and the adjusted p-values using the Benjamini-Hochberg procedure.

not using Octopull. For each tag, we perform a Mann-Whitney-Wilcoxon test [38] on these two groups to determine whether their distribution differs significantly. We also determine the effect size using Cohen's r measure [24], defined as $r = z/\sqrt{N}$ where z is the test statistic from the test and N is the number of observations. Because we are testing multiple null hypotheses, we adjust the obtained p-values using the Benjamini-Hochberg [18] procedure to reduce the probability of false rejections. The results can be seen in Table 5.2.

We see that based on the p-values, there is a significant difference for all tags except 'user-interface' and 'planning', at a significance level of 5%. Based on this, we would reject the null hypothesis. Based on the adjusted values with a treshold of $q = 0.1$, we arrive at the same conclusion. Finally, we consider the strength of these significant correlations using the Cohen's r measure for effect size. We consider a value of $r < 0.2$ as a neglegible effect, and a value of $0.2 < r < 0.5$ as a small effect [24]. We see then that the tag 'review' has a significant correlation with the usage of Octopull, with a small effect size.

This significant difference for the 'review' tag leads us to reject hypothesis $H2_0$: *There is no significant difference in the distribution of tags between Pull Requests using Octopull and Pull Requests not using Octopull.* in favor of the alternative $H2_A$: *There is a significant difference in the distribution of tags between Pull Requests using Octopull and Pull Requests not using Octopull.*.

Takeaway Even though we reject the null hypothesis, we are not convinced of the effect of Octopull on the content of the discussion. Because the results are only significant for

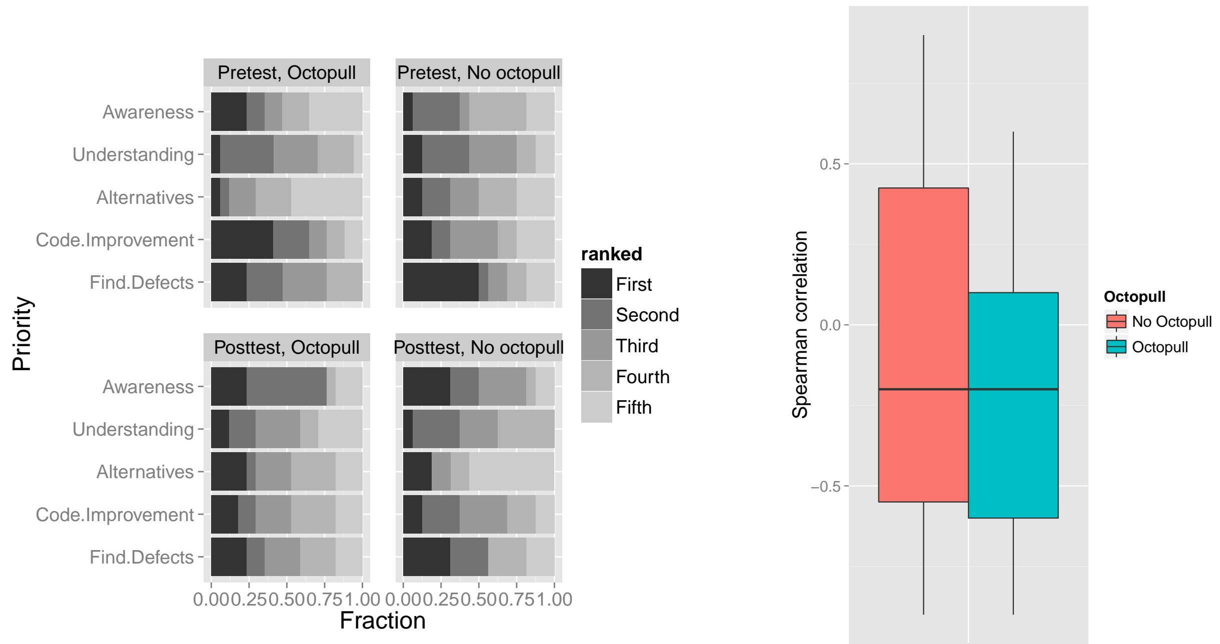

Figure 5.3: Code review priorities from pretest to posttest.
Left: Proportions of code review priorities for pre- and posttest, divided by Octopull and non-Octopull users.
Right: The distribution of spearman correlations between pre- and posttest priorities per user, divided by Octopull and non-Octopull users.

a single tag, we suspect that an alternative explanation is more likely, for example that Octopull users were more diligent in their tagging.

5.2.3 The effect on priorities

RQ4: What is the effect of displaying warnings on the developers' perceived priorities during code review?

To determine if there is a relation between using Octopull and the perceived priorities of code review, we asked students to rank the five most common reasons to do code review. These rankings can be seen in Figure 5.3 on the left. We then determine how much these priorities have changed between the pre- and posttest by taking the Spearman [38] correlation between the rankings, which gives us a number between -1 and 1, where numbers closer to 1 indicate a more similar ranking.

We then compare these correlation coefficients to determine if there is a significant relation with the choice of Octopull. Figure 5.3 shows these values on the right. Visually,

we see that the correlation coefficient tends to be lower for Octopull users (mean -0.20) in comparison to non-Octopull users (mean -0.11). However, a Mann-Whitney-Wilcoxon test [38] on these two samples yields a p-value of 0.786, so we conclude that this difference is not significant.

Because the difference in rankings does not show a significant correlation, we can not reject the null hypothesis $H3_0$: *There is no significant difference in the change of code review priorities from pretest to posttest between Octopull users and non-Octopull users.*

Takeaway We can not reject the null hypothesis and can thus not conclusively determine whether a relation between using Octopull and the developers' perceived priorities of code review exists.

5.3 Threats to validity

In Section 3.1.2 we discussed the threats to validity of the experiment. In this section, we investigate the dataset in an attempt to find such threats.

The students were allowed to self-select into treatment and control groups, and because of this it is possible that a selection bias is present. We investigate several parameters in the dataset to get an indication of the extent of this bias.

The null-hypothesis we formulated about the dataset bias is
Hx_0: *There is no significant difference in the indicated experience on the pretest between Octopull users and non-Octopull users.*

Professional experience	No octopull	Octopull
Yes	20	17
No	77	46

$\chi^2 = 0.54931,\ df = 1,\ p = 0.4586$

Open source contributor	No octopull	Octopull
Yes	18	7
No	79	56

$\chi^2 = 1.0909,\ df = 1,\ p = 0.2963$

Figure 5.4: Contingency tables of Octopull usage versus Professional Software Engineering experience (*left*) and Octopull usage versus Open Source Contribution (*right*)

Every student who completed the pretest indicated whether or not they had experience with professional software development, and whether or not they had experience with contributing to open source projects. Furthermore, we can retrieve a list of GitHub usernames that have logged into the Octopull server.

We created a contingency table of whether students installed Octopull versus whether students have done professional software development in the past. Similarly, we created a contingency table of whether students installed Octopull versus whether students have contributed to open source projects before. These contingency tables are found in Figure 5.4, along with the results of Chi-Square tests on these tables. Based on these Chi-Square tests, we see no significant reason to reject the null hypotheses that

the choice for Octopull and experience with professional software development or open source contributions are independent ($p = 0.46$ and $p = 0.30$ respectively).

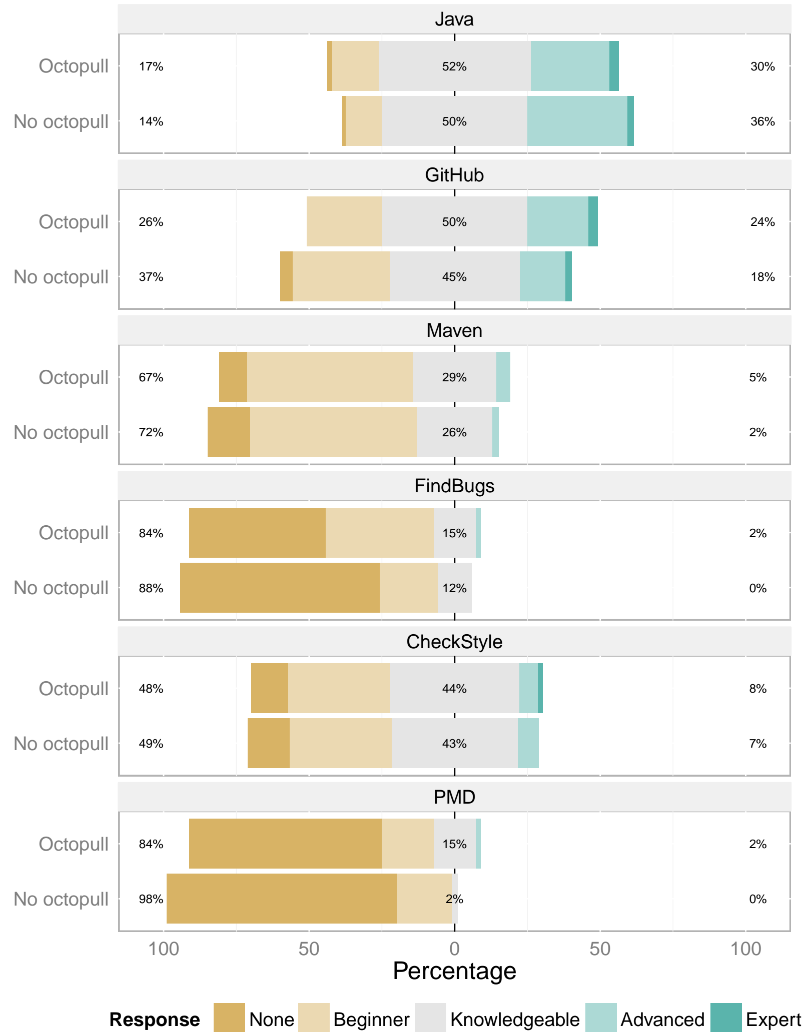

Figure 5.5: Bar-graphs showing the percentage of students that indicated a certain experience level for each tool, split by Octopull and non-Octopull users.

Next, we looked at the reported experience of the students in the pretest. Each student reported their experience with Java, GitHub, Maven, FindBugs, CheckStyle and PMD on a Likert scale [45] ('None', 'Beginner', 'Knowledgeable', 'Advanced' or 'Expert'). The responses are displayed as a bar chart in Figure 5.5. From this chart, we see that those who installed Octopull tend to rate themselves as more experienced with each of the tools, but rate themselves lower on their knowledge of Java.

To discover if there is indeed a significant relation between installing Octopull and

	Java	GitHub	Maven	FindBugs	CheckStyle	PMD
p-value	0.818	0.385	0.617	0.021	0.880	0.008
adjusted value	0.880	0.770	0.880	0.063	0.880	0.050

Table 5.3: The p-values and values adjusted by the Benjamini-Hochberg procedure for Fisher's exact test of independence of Likert scale ratings versus Octopull usage.

experience with these tools, we transform this data to a contingency table for each tool. Because the expected values in these tables are low, the Chi-Square test can not be applied and the Fisher's exact test is used instead. Because we are testing multiple hypotheses, we apply the Benjamini-Hochberg procedure [18] to reduce the probability of false rejections. The p-values and adjusted values can be found in Table 5.3.

Based on the p-values, we would reject the null-hypothesis of independence for FindBugs and PMD (i.e. installing Octopull is unrelated to experience with FindBugs or PMD) at a significance level of 5%. For the adjusted values, with a treshold of $q = 0.1$ we reject the hypotheses for FindBugs and PMD as well. Based on this, we conclude that there is a significant correlation between Octopull installation and experience with FindBugs and PMD. Thus, we reject Hx_0 in favor of the alternative hypothesis Hx_A: *There is a significant difference in the indicated experience between Octopull users and non-Octopull users.*

Takeaway We see a bias in the dataset, as a significant relation between the students' choice to install Octopull and their indicated experience with FindBugs and PMD was established. However, we see no significant reason to assume a relation between the other factors and Octopull.

5.4 Discussion

Based on the collected data, we've sought to answer each of our research questions. However, based on our collected data, we were not able determine whether the tool had an effect. Based on the answers of the students to the open questions on the posttest, we can see that there are some very positive and some very negative reactions to the tool. It would be very interesting to study what strategies developers employ to comprehend the code under review, similar to [53], and what tools are a good fit for each strategy.

Chapter 6

Conclusions and Future Work

6.1 Contributions

The main contribution of this thesis is the Octopull tool. This tool consists of a server and a client component, that work together to provide a visualization of warnings on Pull Requests on GitHub. The client component is currently an extension specific to the Chrome browser. We do not feel this is a major drawback, as the market share of Chrome is large. Additionally, the client can be ported to different browsers with relative ease, if they support the injection of custom JavaScript code. While the tool currently targets Java projects using CheckStyle, FindBugs and PMD as static analysis tools, this is not a technical restriction and the modular architecture of the server ensures that it can be easily extended.

The main aim of creating such a tool was to answer the following question:

> *"Does integrating Static Analysis with Code Review increase the effectiveness of the Code Review process?"*

To answer this question we tested our tool with students of the Software Engineering Methods course. Because we felt it was not right to force students to use the tool, the experiment was designed as a quasi-experiment, i.e. students were non-randomly allocated to the treatment and control group through self-selection.

Under these limitations, we analyzed the results of the experiment in four main areas: developer interest and usage, the effect of the tool on resolved warnings, the effect of the tool on discussion and the effect of the tool on code review priorities. We concluded that a small amount of students were interested in using the tool. Additionally, we concluded that there was no significant evidence of an effect of the tool on resolved warnings, discussion or priorities.

Based on our findings in these four areas, we conclude that we can not be sure whether integrating Static Analysis with Code Review increases the effectiveness of the Code Review process.

6.2 Future Research

As the experiment did not yield statistically significant results, we believe a larger study with more participants will make it possible to determine significant results. Additionally, we believe it would be best to target professional developers and/or open source developers, instead of students, as undergraduate students are not representative of the developer popuplation. If possible, applying randomization across treatment and control group will improve the generalizability of the result.

To increase the target audience, the tool could be made to support different browsers. Additionally, the server application was designed to be extensible to different contexts: static analysis tools and build systems can be replaced by different ones with ease, so that the tool can be studied for almost any GitHub project. This was done to allow the experiment to be repeatable in different companies and organisations. While our tool was created specifically for GitHub, the concept of enriching webpages could be applied to other online code reviewing environments such as Gerrit as well.

Bibliography

[1] Checkstyle. http://checkstyle.sourceforge.net/. Last accessed on 10-12-2015.

[2] Diff utility - wikipedia. https://en.wikipedia.org/wiki/Diff_utility. Last accessed on 10-12-2015.

[3] Eclipse egit github connector. https://github.com/eclipse/egit-github. Last accessed on 12-12-2015.

[4] Gerrit code review. https://www.gerritcodereview.com/. Last accessed on 10-12-2015.

[5] Jersey - restful web services in java. https://jersey.java.net/. Last accessed on 12-12-2015.

[6] Pmd. https://pmd.github.io/. Last accessed on 10-12-2015.

[7] Software engineering methods. http://www.studiegids.tudelft.nl/a101_displayCourse.do?course_id=36658. Last accessed on 12-12-2015.

[8] Surveygizmo. http://surveygizmo.com/. Last accessed on 11-12-2015.

[9] Usage share of web browsers - wikipedia. https://en.wikipedia.org/wiki/Usage_share_of_web_browsers. Last accessed on 12-12-2015.

[10] Tim Ambler and Nicholas Cloud. Browserify. In *JavaScript Frameworks for Modern Web Dev*, pages 101–120. Springer, 2015.

[11] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[12] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[14] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[15] Christian Bauer and Gavin King. Hibernate in action. 2005.

[16] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[17] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

[18] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[19] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[20] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, (10):1294–1304, 1989.

[21] Amiangshu Bosu and Jeffrey C Carver. Peer code review in open source communities using reviewboard. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pages 17–24. ACM, 2012.

[22] Alistair Cockburn. Hexagonal architecture: Ports and adapters ("object structural"). june 19, 2008.

[23] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming.

[24] Jacob Cohen. *Statistical power analysis for the behavioral sciences.* Academic press, 2013.

[25] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[26] Modules CommonJS. Packages, and the sdk, 2011.

[27] Gergely Daróczi. *Mastering Data Analysis with R.* Packt Publishing, 9 2015.

[28] Frederik Michel Dekking. *A Modern Introduction to Probability and Statistics: Understanding why and how.* Springer Science & Business Media, 2005.

[29] Michael Dyer. Verification based inspection. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume 2, pages 418–427. IEEE, 1992.

[30] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[31] ME Fagan. Design and code inspections to reduce errors in program development, IBM systems journal., vol. 15, 1976.

[32] ME Fagan. Advances in software inspections, IEEE trans. software eng., vol. 12, 1986.

[33] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, (4):8–17, 2013.

[34] Eclipse Foundation. Jetty - servlet engine and http server. `https://www.eclipse.org/jetty/`. Last accessed on 12-12-2015.

[35] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, 2006.

[36] 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.

[37] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification.* Cengage Learning EMEA, 2008.

[38] James J Higgins. Introduction to modern nonparametric statistics. 2003.

[39] Gerard J Holzmann. Scrub: a tool for code reviews. *Innovations in Systems and Software Engineering*, 6(4):311–318, 2010.

[40] Philip M Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering*, 3(1):9–35, 1998.

[41] Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API.* Apress, 2006.

[42] Niall Kennedy. Google mondrian: web-based code review and storage, 2006.

[43] John C Knight and E Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, 1993.

[44] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[45] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[46] Mika V Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[47] Johnny Martin and Wei Tek Tsai. N-fold inspection: A requirements analysis technique. *Communications of the ACM*, 33(2):225–232, 1990.

[48] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.

[49] Tim O'Brien and Mountain View Sonatype Inc. *Maven: the definitive guide*. O'Reilly, 2008.

[50] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 161–170. IEEE, 2015.

[51] David L Parnas and David M Weiss. Active design reviews: principles and practices. In *Proceedings of the 8th international conference on Software engineering*, pages 132–136. IEEE Computer Society Press, 1985.

[52] Jennifer Niederst Robbins. *HTML5 Pocket Reference*. " O'Reilly Media, Inc.", 2013.

[53] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, pages 255–265. IEEE Press, 2012.

[54] William R.. Shadish, Thomas D Cook, and Donald Thomas Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Wadsworth Cengage learning, 2002.

[55] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 504–507. IEEE, 2011.

[56] Linus Torvalds and Junio Hamano. Git: Fast version control system. *URL http://git-scm.com*, 2010.

[57] Lawrence G. Votta, Jr. Does every inspection need a meeting? In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '93, pages 107–114, New York, NY, USA, 1993. ACM.

[58] Michal Young and Mauro Pezze. Software testing and analysis: Process, principles and techniques. 2008.

Appendix A

Rubrics

This appendix contains the rubrics used to grade the project work as published on BlackBoard. There are four rubrics: one for the working version of the product after each sprint, one for the final version of the product, one for code quality and one for the management of the sprints. Each is listed below.

A.1 Working Version Rubric

		Exemplary (10, 9, 8)	Competent (7, 6, 5)	Developing (4, 3, 2, 1)	Weight
Requirements	**Specification**	Requirements cover all the core concepts of the game (both functional and non-functional). Requirements are neither contradictory nor ambiguous. All requirements are verifiable and prioritized. Language is clear. The TA's feedback is followed.	Some core concepts of the game are not well covered. Most requirements are neither contradictory nor ambiguous. Most requirements are verifiable and prioritized. Language is sometimes unclear. The TA's feedback is not entirely followed.	Relevant core concepts of the game are not covered. Some requirements are contradictory or ambiguous, and not verifiable. Requirements are not prioritized. Language is unclear. The TA's feedback is mostly not followed.	**15%**
	Implementation	Nearly all requirements are implemented. The reason for not implementing some is clear and motivated.	Only the very core/basic requirements are implemented. The reason for not implementing some is not well motivated.	The implementation does not cover the core requirements of the game. The reason for not implementing certain requirements are unclear and not motivated.	**50%**
Code readability	**Formatting**	Code is well formatted and follows the language’s conventions.	Code is well formatted yet diverges from the language’s conventions with little motivation.	Code is poorly formatted and does not follow the language’s conventions.	**1%**
	Naming	Appropriate and clear names are used for variables, methods, classes, etc.	Most names are clear and appropriate; some names are ambiguous or vague.	Names are vague, ambiguous, or unrelated to their contexts.	**2%**
	Comments	Appropriate use is made of comments to document the code correctly.	Most comments are appropriately used yet some lack purpose or are unnecessary.	Little to no comments of value; code is poorly documented via comments.	**2%**
Continuous integration	**Building**	Almost all failing commits are quickly fixed. Testing is done correctly with every build.	The most important failing commits are fixed in not too long time. Testing is done correctly with most builds.	Failing commits are not fixed in reasonable time. Testing is poorly integrated with builds.	**5%**
	Testing	The system has a >75% of meaningful test coverage (maven cobertura, line coverage). Testing reports.	The system has a >45% of meaningful test coverage.	The system has a <20% of meaningful test coverage.	**15%**
Pull-based development model		All new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests. Pull-requests are thoroughly reviewed also considering the output of tools used in the continuous integration.	Not all new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests. Pull-requests are superficially reviewed and the output of tools used in the continuous integration partially considered.	Very few/no new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests. Pull-requests are rarely reviewed and the output of tools used in the continuous integration almost not considered.	**5%**
Tooling (i.e., GitHubm Cobertura, FindBugs, PMD, CheckStyle, Octopull)		The required tools are correcly setup and used.	The required tools are correctly setup but not correcly used.	The required tools are neither correctly setup nor correctly used.	**5%**

A.2 Final Version Rubric

		Exemplary			Competent			Developing				Weight
		10	9	8	7	6	5	4	3	2	1	
Completeness		All the main requirements are met and additional optional features are added			Almost all the main requirements are met			A few of the main requirements are met				**25%**
Source code quality		Design patterns are well placed and implemented. Methods are of a proper size. No design flaws are present. Other software engineering principles are also used appropriately and correctly.			Design patterns are well placed, though not implemented completely. Only few and limited design flaws are present. Other software engineering principles are also used appropriately yet sometimes incorrectly.			Inappropriate use is made of design patterns or anti-patterns are used. Design flaws (e.g., God classes and feature envy methods) are readily present. Other software engineering principles are also used inappropriately or not all.				**35%**
Software architecture		The system has a clear over-arching software architecture.			The system has a vague over-arching software architecture.			The system has no clear over-arching standard software architecture.				**12%**
Testing		The system has a >75% of meaningful test coverage (maven cobertura, line coverage).			The system has a >45% of meaningful test coverage.			The system has a <20% of meaningful test coverage.				**20%**
Code readability	**Formatting**	Code is well formatted and follows the language's conventions.			Code is well formatted yet diverges from the language's conventions with little motivation.			Code is poorly formatted and does not follow the language's conventions.				**2%**
	Naming	Appropriate and clear names are used for variables, methods, classes, etc.			Most names are clear and appropriate; some names are ambiguous or vague.			Names are vague, ambiguous, or unrelated to their contexts.				**3%**
	Comments	Appropriate use is made of comments to document the code correctly.			Most comments are appropriately used yet some lack purpose or are unnecessary.			Little to no comments of value; code is poorly documented via comments.				**3%**

A.3 Quality Rubric

		Exemplary			Competent			Developing				Weight
		10	9	8	7	6	5	4	3	2	1	
Code quality		The code base follows sound design principles and object-oriented programming.			The code base generally follows sound design principles and object-oriented programming, but there are lower quality parts.			The code base does not follow sound design principles and object-oriented programming and there are glaring mistakes.				**20%**
Code readability	**Formatting**	Code is well formatted and follows the language's conventions.			Code is well formatted yet diverges from the language's conventions with little motivation.			Code is poorly formatted and does not follow the language's conventions.				**2%**
	Naming	Appropriate and clear names are used for variables, methods, classes, etc.			Most names are clear and appropriate; some names are ambiguous or vague.			Names are vague, ambiguous, or unrelated to their contexts.				**5%**
	Comments	Appropriate use is made of comments to document the code correctly.			Most comments are appropriately used yet some lack purpose or are unnecessary.			Little to no comments of value; code is poorly documented via comments.				**3%**
Continuous integration	**Building**	Almost all failing commits are quickly fixed.			The most important failing commits are fixed in not too long time.			Failing commits are not fixed in reasonable time.				**5%**
	Testing	The system has a >75% of meaningful test coverage (maven cobertura, line coverage). A testing document is available to describe special cases.			The system has a >45% of meaningful test coverage.			The system has a <20% of meaningful test coverage.				**30%**
Tagging		All new commit messages and pull-request/review comments are properly tagged.			Not all new commit messages and pull-request/review comments are properly tagged.			Very few/no new commit messages and pull-request/review comments are properly tagged.				**5%**
Tooling (i.e., GitHubm Cobertura, FindBugs, PMD, CheckStyle, Octopull)		The required tools are correcly setup and used.			The required tools are correctly setup but not correcly used.			The required tools are neither correctly setup nor correctly used.				**5%**
Pull-based development	**Branching**	All new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests.			Not all new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests.			Very few/no new features, bug fixes, etc. are done in a separate branch and integrated through pull-requests.				**10%**
	Code review	Pull-requests are thoroughly reviewed also considering the output of tools used in the continuous integration.Testing reports.			Pull-requests are not thoroughly reviewed and the output of tools used in the continuous integration partially considered.			Pull-requests are poorly reviewed and the output of tools used in the continuous integration almost not considered.				**15%**

A.4 Sprint Management Rubric

		Exemplary (10 / 9 / 8)	Competent (7 / 6 / 5)	Developing (4 / 3 / 2 / 1)	Weight
Tasks	Definition	Tasks are: clearly defined, have a reasonable granularity, and have a clear definition of done	Not all tasks are: clearly defined, have a reasonable granularity, or have a clear definition of done.	Most tasks are: not clearly defined, do not have a reasonable granularity, and do not have a clear definition of done.	20%
	Splitting	Tasks are evenly split among group members.	Tasks are not evenly split among group members.	Tasks are unevenly divided among group members.	15%
	Responsibility	A responsible person is clearly defined for each task.	A responsible for each task is defined but sometimes vaguely.	Responsibilities are loosely defined.	10%
Learning from History	Estimation	Estimation is realistically based on the experience from the previous Sprint(s).	Estimation is based on the previous Sprint(s), yet it is still unrealistic for some tasks.	Estimation is not based on the previous Sprint(s) and is mostly unrealistic.	15%
	Prioritization	Prioritization of tasks is based on the experience from the previous Sprint and is well motivated.	Prioritization of tasks is mostly based on the experience of the previous Sprint yet is vaguely motivated.	Prioritization of tasks is not based on the experience from the previous Sprint and vaguely motivated.	15%
	Reflection	Reflection coincides with the data of the last Sprint(s) and makes motivated adjustments for the next Sprint.	Reflection coincides with the actual data provided, but only vaguely describes adjustments needed for the next Sprint.	Reflection does not coincide with the actual data provided and makes no meaningful adjustments to the process.	25%

Appendix B

Pretest Questionnaire

The pretest questionnaire was completed by all students participating in the course. The questionnaire was created using SurveyGizmo [8]. The questions are enumerated below. For questions with answers that have no specific ordering, the order was randomized.

1. What is your age? ___
2. What is your NetID? _________
   Note: This is only used for administrative purposes. The answers on this survey will not impact your grade in any way.
3. What is your username on GitHub? _________
4. Have you taken this course before? Yes/No.
5. *If previous answer was yes,* In what years did you take the course before?
   - 2014-2015
   - 2013-2014
   - 2012-2013
   - 2011-2012
   - Other: _________
6. Do you have experience with paid software engineering work?
   - Yes, I have worked in software engineering for a year or more.
     Please specify how many _________
   - Yes, I have worked in software engineering for less than a year.
   - No, I have not worked in software engineering.
7. Have you ever contributed to Open Source projects? Yes/No.

8. *If previous answer was yes,* To what Open Source projects have you contributed? _________

9. What is your username, if any, on the following websites:

   - BitBucket: _________
   - StackOverflow: _________
   - GitLab: _________
   - SourceForge: _________
   - CodePlex: _________

   The following question uses this scale to rate your experience:

   1 - None (you don't know this subject);
   2 - Beginner (you are familiar with this subject but still have some difficulties to use it);
   3 - Knowledgeable (you are comfortable in this subject and currently use it daily);
   4 - Advanced (you currently consider yourself highly proficient in this subject);
   5 - Expert (your colleagues look for you when they need help in this subject, and you feel confident to help them).

10. What is your experience with _________

    - Java
    - GitHub
    - Pull Requests
    - Static Analysis Tools
    - Maven
    - FindBugs
    - CheckStyle
    - PMD
    - Continuous Integration
    - Travis CI
    - Code Review
    - Working in a development team

    The following question uses this scale to rate your usage:

    a) Not at all
    b) Rarely (less than 1 day a week on average)

c) Sometimes (1 or 2 days a week on average)
d) Often (3 or 4 days a week on average)
e) Always (at least 5 days a week on average)

11. How often have you used _________ in the last 6 months?
    - Java
    - GitHub
    - Pull Requests
    - Static Analysis Tools
    - Maven
    - FindBugs
    - CheckStyle
    - PMD
    - Continuous Integration
    - Travis CI
    - Code Review
    - Working in a development team

12. What is in your opinion the priority of code review to software engineering?

    ◯ Not a priority
    ◯ Low priority
    ◯ Medium priority
    ◯ High priority
    ◯ Essential

13. Please rank the following from most important to least important reason to do code review:
    - Finding defects or bugs.
    - Improving the code style or quality.
    - Think of other ways to implement the feature or solve the problem.
    - Understand the code of your colleagues.
    - Know what your colleagues are working on.

14. According to your experience, rank these outcomes of code review from most to least common:
    - Finding defects or bugs.

- Improving the code style or quality.
- Think of other ways to implement the feature or solve the problem.
- Understand the code of your colleagues.
- Know what your colleagues are working on.

15. What is in your opinion the priority of static analysis to the software engineering process?

    ◯ Not a priority
    ◯ Low priority
    ◯ Medium priority
    ◯ High priority
    ◯ Essential

16. What are in your opinion the most important reasons to use static analysis tools?

    ____

17. What means of communication have you used when working on a development team?

    ☐ Face to face communication
    ☐ Video Voice calling
    ☐ Real-time chat (e.g. Skype)
    ☐ Asynchronous means (e.g. email)
    ☐ Asynchronous comments (e.g. leaving a comment on GitHub issues)
    ☐ Documentation
    ☐ Issue tracking system
    ☐ Multi-person chat (e.g. IRC)
    ☐ Other: _________

18. Please rank the following means of communication with a development team from most to least effective for you:

    - Face to face communication
    - Video Voice calling
    - Real-time chat (e.g. Skype)
    - Asynchronous means (e.g. email)
    - Asynchronous comments (e.g. leaving a comment on GitHub issues)
    - Documentation
    - Issue tracking system
    - Multi-person chat (e.g. IRC)

Appendix C

Posttest Questionnaire

All students participating in the course were asked to complete the posttest questionnaire. The questionnaire was created using Google Forms. The questions are enumerated below. For questions with answers that have no specific ordering, the order was randomized.

1. What is your username on GitHub? _________

   The following question uses this scale to rate your experience:

   1 - None (you don't know this subject);

   2 - Beginner (you are familiar with this subject but still have some difficulties to use it);

   3 - Knowledgeable (you are comfortable in this subject and currently use it daily);

   4 - Advanced (you currently consider yourself highly proficient in this subject);

   5 - Expert (your colleagues look for you when they need help in this subject, and you feel confident to help them).

2. What is your experience with _________

   - Java
   - GitHub
   - Pull Requests
   - Static Analysis Tools
   - Maven
   - FindBugs
   - CheckStyle
   - PMD
   - Continuous Integration

- Travis CI
- Code Review
- Working in a development team

3. What is in your opinion the priority of code review to software engineering?

   ◯ Not a priority
   ◯ Low priority
   ◯ Medium priority
   ◯ High priority
   ◯ Essential

4. Please rank the following from most important to least important reason to do code review:

   - Finding defects or bugs.
   - Improving the code style or quality.
   - Think of other ways to implement the feature or solve the problem.
   - Understand the code of your colleagues.
   - Know what your colleagues are working on.

5. What is in your opinion the priority of static analysis to the software engineering process?

   ◯ Not a priority
   ◯ Low priority
   ◯ Medium priority
   ◯ High priority
   ◯ Essential

6. Have you used Octopull during the project? Yes/No.

7. *If the student did not use Octopull.* What were your main reasons to decide against using Octopull?

   ____

8. *If the student used Octopull.* What were your main reasons to use Octopull?

   ____

9. *If the student used Octopull.* Were you satisfied with the Octopull tool?

   ◯ Very dissatisfied

- ◯ Dissatisfied
- ◯ Unsure
- ◯ Satisfied
- ◯ Very satisfied

10. *If the student used Octopull.* What would you like to see improved in Octopull?

    ____

11. *If the student used Octopull.* Would you want to use Octopull again in a future project? Yes/No.

12. *If the student used Octopull.* Do you have any other comments related to Octopull?

    ____