University of Stavanger

**OLE JØRGEN ESPELAND**

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

# Quickfeed Support for Feedback via Pull Requests and Issues

Bachelor's Thesis - Computer Science - May 2022

# Abstract

QuickFeed is an automatic evaluation system, developed at the University of Stavanger. Designed with the intent of easing the submission of assignments, QuickFeed tests student code as they push it to a supported source control management system. Based on the results of these tests, QuickFeed publishes feedback to students via its custom web interface.

In this thesis, we explore the possibility of further expanding how students receive feedback on their submissions, by using GitHub's pull request and issue features.

# Acknowledgements

I would like to thank my supervisor, Professor Hein Meling; not just for providing guidance and feedback throughout the project, but also for reviewing and helping make improvements to my implementation.

I would also like to thank Jostein Hagen Lindhom, similarly for providing guaidance and support, but especially for always being eager and ready to help with any issues I faced, often saving me from much frustration.

# Contents

**Abstract** **ii**

**Acknowledgements** **iii**

**1 Introduction** **2**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Background** **4**

2.1 GitHub . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.1 Issues . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.2 Pull Requests . . . . . . . . . . . . . . . . . . . . 5

2.2 QuickFeed . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2.1 Protocol Buffers . . . . . . . . . . . . . . . . . . . 7

2.2.2 QuickFeed Repository Structure . . . . . . . . . . . . . . . 8

2.2.3 The Score Package . . . . . . . . . . . . . . . . . . . . . . . 10

2.2.4 Testing Assignments . . . . . . . . . . . . . . . . . . . . . . 11

2.2.5 Webhooks . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.2.6 Source Control Management API . . . . . . . . . . . . . . 13

**3 Related Work** **14**

3.1 Earlier attempt . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**4 Approach** **15**

4.1 Student Repository Access . . . . . . . . . . . . . . . . . . . . . 16

4.1.1 Possible Solutions . . . . . . . . . . . . . . . . . . . . . . . 17

4.1.2 Limiting Scope to Group Repositories . . . . . . . . . . . 18

4.2 Creating Pull Requests . . . . . . . . . . . . . . . . . . . . . . . 19

4.2.1 Manual Creation . . . . . . . . . . . . . . . . . . . . . . . . 20

4.2.2 Automatic Creation . . . . . . . . . . . . . . . . . . . . . . 20

4.2.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.3 Issue Creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

4.4 Automated Student Feedback . . . . . . . . . . . . . . . . . . . . 22

4.4.1 Pull Request Comments . . . . . . . . . . . . . . . . . . . . . 23

**5 Implementation** **24**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1.1 Feature Overview . . . . . . . . . . . . . . . . . . . . . . 24

5.2 Existing implementation . . . . . . . . . . . . . . . . . . . . . . 25

5.2.1 SCM Expansion . . . . . . . . . . . . . . . . . . . . . . . 26

5.2.2 Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

5.2.3 Finding and Parsing Tasks . . . . . . . . . . . . . . . . . 26

5.3 Managing Tasks and Issues . . . . . . . . . . . . . . . . . . . . . 28

5.3.1 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

5.3.2 Differentiating Data Objects . . . . . . . . . . . . . . . . 29

5.3.3 Data Structures . . . . . . . . . . . . . . . . . . . . . . . 30

5.3.4 Synchronizing Tasks . . . . . . . . . . . . . . . . . . . . . 31

5.3.5 Synchronizing Issues . . . . . . . . . . . . . . . . . . . . . 33

5.4 Managing Pull Requests . . . . . . . . . . . . . . . . . . . . . . . 34

5.4.1 Pull Request Data Structure . . . . . . . . . . . . . . . . 34

5.4.2 Creating Pull Requests . . . . . . . . . . . . . . . . . . . 35

5.4.3 Closing Pull Requests . . . . . . . . . . . . . . . . . . . . . 36

5.5 Scoring Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

5.6 Manual Feedback . . . . . . . . . . . . . . . . . . . . . . . . . 38

5.6.1 Assigning Reviewers . . . . . . . . . . . . . . . . . . . . . . 39

5.6.2 Approving Pull Requests . . . . . . . . . . . . . . . . . . . 41

5.7 Automatic Feedback . . . . . . . . . . . . . . . . . . . . . . . 41

5.7.1 Formatting Feedback Comments . . . . . . . . . . . . . . . 42

5.7.2 Publishing Feedback Comments . . . . . . . . . . . . . . . 43

**6 Insights** **45**

6.1 User Experience . . . . . . . . . . . . . . . . . . . . . . . . . 45

6.2 Known Limitations . . . . . . . . . . . . . . . . . . . . . . . . 46

6.2.1 Differing Branch Names . . . . . . . . . . . . . . . . . . . 47

6.2.2 Human Error . . . . . . . . . . . . . . . . . . . . . . . . . . 47

6.2.3 Manually Graded Assignments . . . . . . . . . . . . . . . . 48

6.3 Possible Improvements . . . . . . . . . . . . . . . . . . . . . . 48

6.3.1 GitHub App . . . . . . . . . . . . . . . . . . . . . . . . . . 48

6.3.2 Task Naming Format . . . . . . . . . . . . . . . . . . . . . 49

6.3.3 Linking Issues . . . . . . . . . . . . . . . . . . . . . . . . . 50

6.3.4 Support for Regular Assignments . . . . . . . . . . . . . . 50

**7 Future Work** **51**

7.1 Further QuickFeed Integration . . . . . . . . . . . . . . . . . . . . . 51

7.2 Further GitHub Integration . . . . . . . . . . . . . . . . . . . . . . . . 52

**8 Conclusions** **54**

**Bibliography** **55**

# Chapter 1

# Introduction

The University of Stavanger has for the last couple of years been developing a project known as QuickFeed. Since its inception in 2015, both teachers and students have contributed to its development [4, 3, 2, 10].

QuickFeed aims to ease the process of submitting student assignments for both students and the teaching staff. Though not exclusively, the project is primarily developed with the intent of handling code based assignments. QuickFeed will run predetermined tests on submitted code, and uses their results to provide students with feedback via its web interface. Using QuickFeed, assignments can be both scored and graded without the teaching staff ever having to intervene.

To accommodate these features, QuickFeed uses GitHub to, amongst other things, manage student code. GitHub is therefore one of the primary sites students interact with, when working on QuickFeed managed assignments. Because of this, there is a desire to further integrate QuickFeed with GitHub. Specifically, we aim to use GitHub's pull request and issue features to give students another avenue for manual and automatic feedback.

## 1.1 Motivation

In the current iteration of QuickFeed, students only receive automated feedback on their code through QuickFeed's custom web interface. Here, they can see what parts of their code is failing, and also whether they have successfully passed the assignment. The motivation behind this thesis is to further expand on how students receive feedback when working on an assignment.

Using GitHub pull requests we can facilitate manual feedback directly on student code, by using its review features. Pull requests also have the potential to support automatic feedback, e.g., by another GitHub feature called workflows.

## 1.2 Objectives

This project has the following fundamental objectives.

First of, we want to give teachers the option of subdividing assignments into individual tasks, each describing specific problems students need to solve. When a student wants to solve a given assignment task, they create a pull request, in which they implement all code relevant for that task.

Furthermore, in this project, we want QuickFeed to assign reviewers to these pull requests when appropriate. These reviewers can request changes and recommend improvements where they deem necessary, thereby providing students with a new way of receiving feedback.

As students push code to their pull requests, we also want QuickFeed to publish some sort of automatic feedback, giving the students a general indication on how well they are doing, directly on the pull requests themselves.

# Chapter 2

# Background

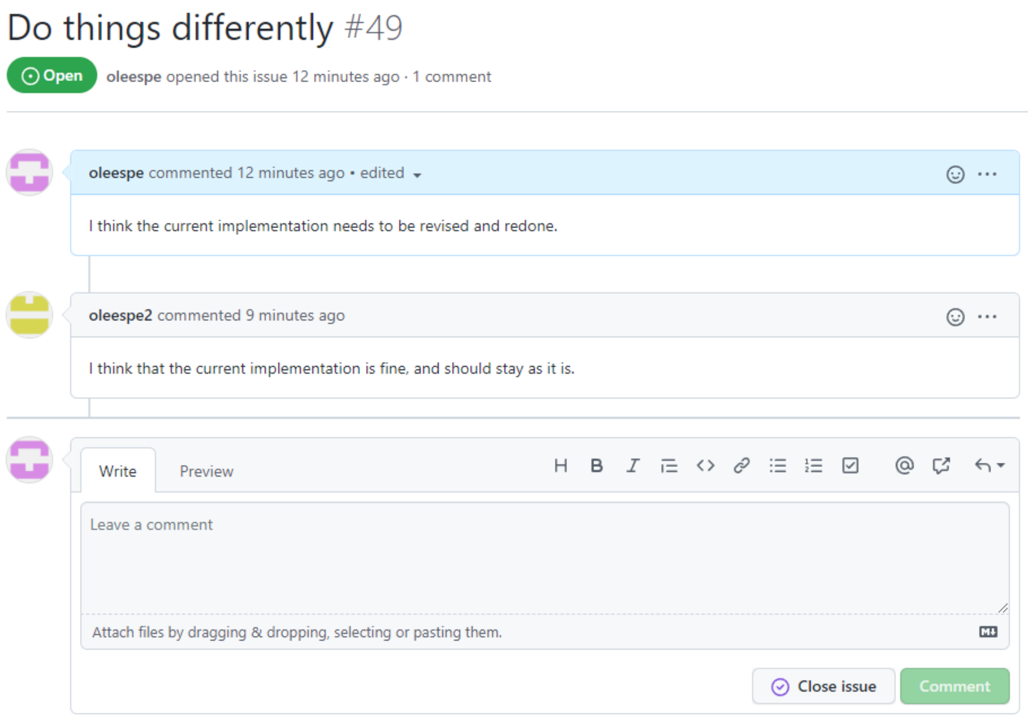
In this chapter we describe existing technology and concepts that will be referred to throughout the thesis.

## 2.1 GitHub

QuickFeed already relies on GitHub to manage assignments and student code. This section looks at certain GitHub features that we want QuickFeed to support.

### 2.1.1 Issues

GitHub issues is a useful feature for tracking, discussing and logging various issues/problems on a GitHub repository. Any collaborator to a repository may open a GitHub issue, and describe any problem, idea or issue they have. Other users may then contribute to the issue by commenting on it, thereby easing the communication process within a project. GitHub issues can therefore function as a central discussion hub, which makes it especially useful for larger projects involving several people.


Do things differently #49

Open oleespe opened this issue 12 minutes ago · 1 comment

oleespe commented 12 minutes ago • edited

I think the current implementation needs to be revised and redone.

oleespe2 commented 9 minutes ago

I think that the current implementation is fine, and should stay as it is.

Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting or pasting them.
Close issue
Comment

Figure 2.1: Example of a GitHub issue comment section

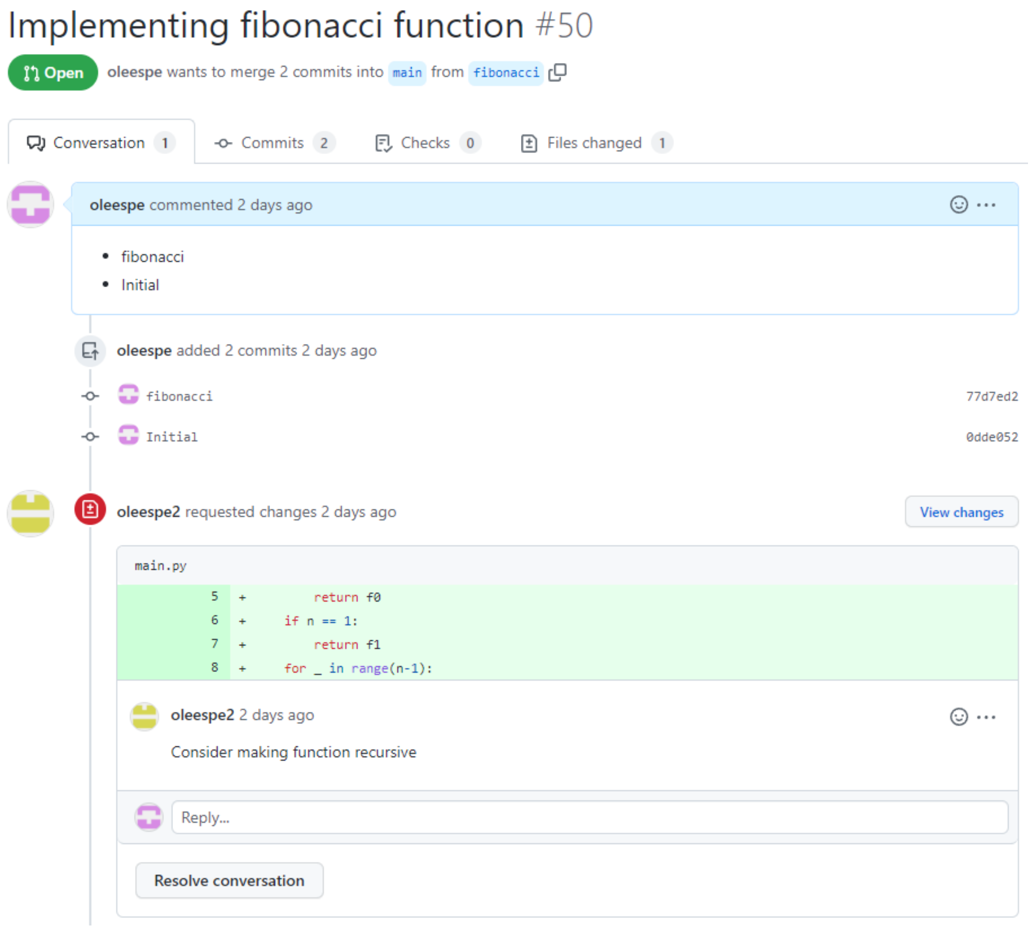
### 2.1.2 Pull Requests

Pull requests are a desire to merge any feature branch, into the main branch of a git repository. GitHub supports managing pull requests through its user interface, and allows any contributor to a repository to create a pull request. Through the user interface, progress on a branch can be tracked, reviewed and commented on. In this sense, GitHub pull requests function as a central hub for feature branches.

Code review is also an important part of the pull request process. Any eligible user may review, comment on, and request changes to the source code of a given pull request. The reviewer may also use reviews to approve a pull request for merging.

Pull requests can also be linked to issues. Doing this will automatically associate any linked issue to the pull request, and cause them to close when the pull request closes. A common workflow would be to create an issue, describing a problem, and then creating an associated pull request for this issue.

Through its features, GitHub pull requests provide an efficient and manageable way of implementing new features to any project.


Implementing fibonacci function #50

oleespe wants to merge 2 commits into main from fibonacci

oleespe commented 2 days ago

- fibonacci
- Initial

oleespe added 2 commits 2 days ago

```
fibonacci    77d7ed2
Initial      0dde052
```

oleespe2 requested changes 2 days ago

```
main.py
5 +         return f0
6 +     if n == 1:
7 +         return f1
8 +     for _ in range(n-1):
```

oleespe2 2 days ago

Consider making function recursive

Figure 2.2: Example of a GitHub pull request

## 2.2 QuickFeed

QuickFeed provides two primary features. A web interface that allows students to enroll in courses, create groups and receive feedback on their submissions. And a backend that tests, scores and grades student submitted code. The backend is primarily implemented with the Go programming language, while the frontend uses TypeScript and the React library. For data storage, QuickFeed relies on an SqLite-database, managed using the the GORM library.

In this section we further describe some of QuickFeed's key concepts, as well as the technology it uses.

### 2.2.1 Protocol Buffers

Protocol buffers are a language-neutral mechanism for serializing structured data [8]. It is often abbreviated as protobuf, and is used to define messages in a *.proto* file. When the file is compiled, data structures and methods for the desired language are generated in a separate file. QuickFeed uses protobuf to generate most of its data structures. These are then stored in its internal database when necessary.

```
 1 message Assignment {
 2     uint64 ID                        = 1;
 3     uint64 CourseID                  = 2;
 4     string name                      = 3;
 5     string scriptFile                = 4;
 6     string deadline                  = 5;
 7     bool autoApprove                 = 6;
 8     uint32 order                     = 7;
 9     bool isGroupLab                  = 8;
10     uint32 scoreLimit                = 9;
11     uint32 reviewers                 = 10;
12     uint32 containerTimeout          = 11;
13     repeated Submission submissions = 12;
14     repeated GradingBenchmark gradingBenchmarks = 14;
15 }
```

Code 2.1: Assignment message

Code 2.1 is an example of such a message. It holds a reference to a single **Course** message and several **Submission** messages, as seen on Line 3 and 13 respectively. When compiled, the resulting data structure is used by QuickFeed to represent assignments.

QuickFeed also uses protobuf messages in conjunction with gRPC, to facilitate server-client communication. This part however, is not relevant for this project.

### 2.2.2 QuickFeed Repository Structure

When a teacher creates a course in QuickFeed, a GitHub organization is created to represent it. Within this organization, three initial repositories are created, as well as student repositories when students enroll. They are all described as follows.

*info*: This repository simply holds information about a course. The repository is available to all students, and would typically work as a simple information hub. It is created and managed by the teaching staff.

*assignments*: The repository responsible for presenting the assignments to students. Every assignment in a course is represented as a unique folder within this repository. As teachers push new assignments to this repository, or update existing ones, students pull the changes to their own local git repositories.

*tests*: Teachers use this repository to store both test code and files needed to facilitate automatic testing of student code. It's folder structure should be one-to-one with *assignments*, with every assignment being represented by a unique folder. When students submit code for an assignment, QuickFeed tests it with the test code found in the folder representing that assignment. In addition, the repository contains assignment specific *assignment.yml* files, as well as a *scripts* folder. The *scripts* folder is used to store a *run.sh* file and a Dockerfile.

The *assignment.yml* file is used by teachers to specify assignment specific settings. For instance, it can contain information about an assignment's deadline, and whether it is manually approved.

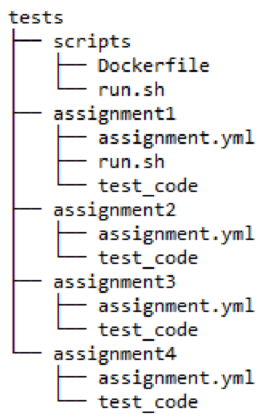
```
assignmentorder: 1
name: "lab1"
scriptfile: "run.sh"
deadline: "15-05-2022T16:00"
autoapprove: false
```

```
isgrouplab: true
reviewers: 2
```

Code 2.2: Example contents of an *assignment.yml* file

The *run.sh* file contains the script used by QuickFeed when testing student code. As explained, it is contained within the *scripts* folder, but can also be located within specific assignments. When running tests on an assignment, QuickFeed will prioritize using any assignment specific script files.

Finally, the Dockerfile contained within *scripts* is used to create a docker container. Within this container, QuickFeed runs the appropriate script, derived from *run.sh*.



```
tests
├── scripts
│   ├── Dockerfile
│   └── run.sh
├── assignment1
│   ├── assignment.yml
│   ├── run.sh
│   └── test_code
├── assignment2
│   ├── assignment.yml
│   └── test_code
├── assignment3
│   ├── assignment.yml
│   └── test_code
└── assignment4
    ├── assignment.yml
    └── test_code
```

Figure 2.3: Example of a tests repository folder structure

Every time a teacher pushes to *tests*, QuickFeed runs the function UpdateFromTestsRepo. This function will parse an assignment's *assignment.yml* file, and use its contents to create data objects defined by the **Assignment** message. They are then used to create or update assignment database records.

Being a repository that is used and managed solely by the teaching staff, it follows that students do not have access to its contents.

Student repositories: There are two types of student repositories, user and group repositories. A user in this case would simply be any student who has enrolled in the course. Whereas a group is any number of students that are working together. Naturally, student repositories are only accessible by the students associated with them, and the teaching staff.

Student repositories are named according to the following format: *name-labs*. For an enrolled student, *name* would be their GitHub user name. A group's *name* however, is defined by the group members themselves when they enroll the group.

These repositories are the ones students push their code to as they work on assignments.

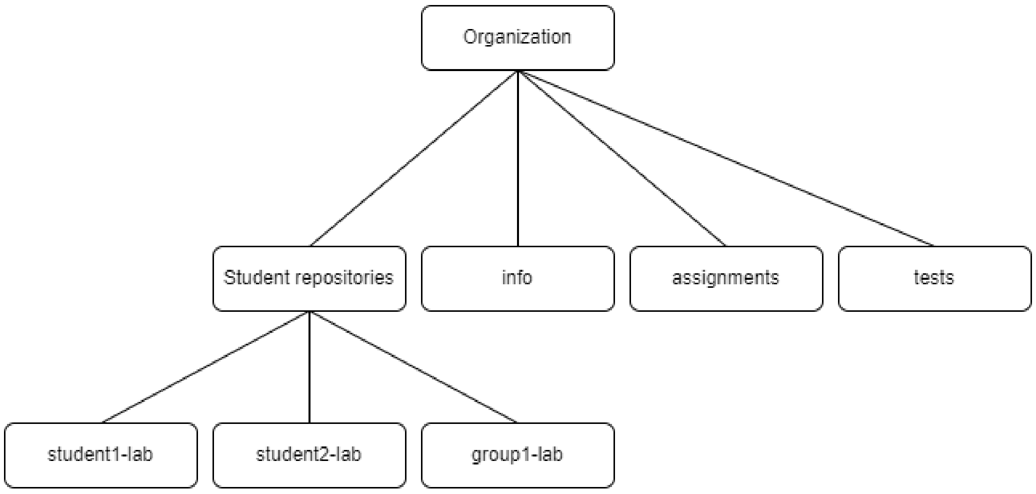


Figure 2.4: GitHub repository structure for a QuickFeed course

### 2.2.3 The Score Package

QuickFeed's score package allows for scoring student submitted code. Every time a student pushes code to their repository, QuickFeed will run tests on the code, and generate a total score ranging from 0 to 100. In this section we describe the parts of this package that are relevant for this thesis.

When teachers develop and push assignments to *tests*, they also create test code that is meant to test student submitted code. As part of the score package, teachers can specify for individual tests, what score they should give, and how that score is gained. For example, a test can loop through several test conditions, and then decrement the score every time it fails. These tests have to be explicitly added by teachers using either of the following methods: Add and AddSub. Doing so, they are included in the pool of all tests that constitute an assignment.

Each individual score is defined by the protobuf message in Code 2.3. When QuickFeed is finished running all tests, a test results data object is extracted containing a list of all these scores.

```
message Score {
    uint64 ID =           1;
    uint64 SubmissionID = 2;
    string Secret =       3;
    string TestName =     4;
    int32 Score =         6;
    int32 MaxScore =      7;
    int32 Weight =        8;
}
```

Code 2.3: Score message

The fields Score and MaxScore are used to represent a given test's relative score, e.g., 5 out of 7. Weight on the other hand, defines the weight of any test in relation to all others in an assignment. If we have a test with a given weight of 5, and an assignment where all test weights summed up give a total weight of 10, our test would account for 50% of the entire assignment. So, if a student gets a score of 3 out of 6 on this test, it would give a score of 25 on the assignment as a whole. Adding up all scores like this for an entire assignment, gives us the assignment total score.

### 2.2.4 Testing Assignments

Having explored how QuickFeed scores student code, this section will further explore parts of the assignment submission process that are important.

As students push code to their GitHub repository, QuickFeed will do two things.

First, it will determine the assignments that have been worked on since the last push. Then, for every worked on assignment, it will run the tests that are defined for that assignment in *tests*.

Running the tests is done by QuickFeed's ci package. To set up a test environment, it uses the Dockerfile found in *tests* to create a docker container. It then continues by running the assignment's script file, as mentioned in Section 2.2.2. These script files can be assignment specific, but generally they use git clone to merge the students code with the assignment tests; followed up by a command to run all tests. The ci package also supports supplying the script with arguments, e.g., to clone the correct student repository.

When this process is finished, QuickFeed extracts the test run's results and uses them to, amongst other things, determine whether the student got a passing score.

### 2.2.5 Webhooks

QuickFeed communicates with GitHub in two ways. In this section we will detail one of them, namely webhooks.

A webhook is a ”user-defined callback over HTTP” [1]. In general, webhooks allow developers to listen to events from any supporting site. When any such event occurs, an HTTP request is sent to the address configured for the webhook. The request contains data about the event, usually in a JSON format.

QuickFeed uses webhooks to retrieve data from push events on a course. When a new course is created, QuickFeed creates a webhook on the GitHub organization of the given course. This webhook is only triggered by push events, which are then handled by QuickFeed accordingly:

- If a push event is associated with the *tests* repository, QuickFeed will update the course assignments.

- If a push event is associated with a student repository, QuickFeed will determine the assignments that have been worked on, and run the assignment specific tests on them.

### 2.2.6 Source Control Management API

The second way QuickFeed communicates with GitHub is through a custom API.

To facilitate communication with potentially any source control management system, a custom source control management API has been developed for QuickFeed. Usually abbreviated as the SCM API, the current iteration of QuickFeed supports interacting with GitHub through this API, via the go-github library. The library itself, communicates with GitHub's own REST API using HTTP requests.

To authenticate with this GitHub API, QuickFeed is implemented as an OAuth app. In essence, this means that QuickFeed can authenticate with the GitHub API, as the identity of any of its users [5].

The SCM API allows QuickFeed to perform various tasks, such as creating or updating repositories, creating webhooks, managing teams and more. QuickFeed's SCM API, together with webhooks, form a 2-way communication stream between QuickFeed and GitHub.

# Chapter 3

# Related Work

## 3.1 Earlier attempt

Adil Khurshid presents an attempt at implementing QuickFeed support for feedback via pull requests and issues [9]. The motivation and desired result for his project is very similar to this one.

Adil's approach can be summarized as follows. To subdivide assignments into tasks, he describes teachers creating task markdown files in *tests*. From the contents of these markdown files, GitHub issues are created on all student repositories. Furthermore, when students want to solve tasks, they create pull requests. When an assignment deadline has passed, student and teacher reviewers are assigned to these pull requests, thereby facilitating feedback on them.

A complete and finished implementation for the above approach was never accomplished, which is why the project is continued here.

# Chapter 4

# Approach

The general approach taken to fulfill the project's goals can be summarized as follows.

To subdivide assignments into tasks, teachers will create and push markdown files to *tests*. Each task has a title and body, which is described by the contents of one of these files. For every task a teacher creates, QuickFeed creates a GitHub issue on all student repositories in a course to represent them. These issues have the same title and body as the task they are based on. This way, issues are used as a representation of tasks accessible to students.

To solve a task, students will create a pull request for the issue that represents that task. When students push code to pull requests, QuickFeed will run tests on it, similarly to the way it currently tests entire assignments. Based on the results from a given test run, QuickFeed publishes feedback on the relevant pull request. Furthermore, when enough tests pass, we assign one teacher and co-student to review the pull request. Only when the teacher approves the pull request, can it be merged.

This general approach can be visualized in Figure 4.1. It shows a single task that serves as the basis for individual issues on three different student repositories.

For each of these issues, a pull request is created to solve the task the issue is based on.

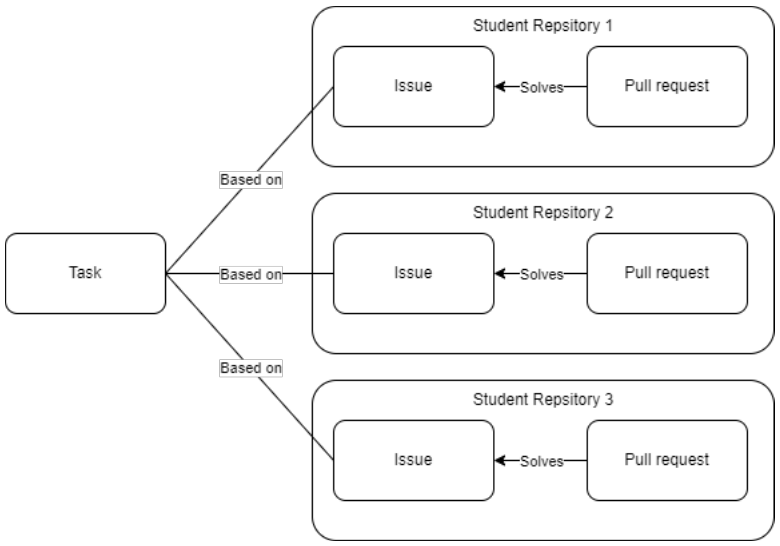


Figure 4.1: General approach

Before we can implementing anything, there are a few lingering design decisions regarding this approach. For instance, how do we publish automatic feedback to a pull request? In this chapter we discuss and analyse these design decisions, and try to give a rationale for the decisions that were made.

## 4.1 Student Repository Access

Since students do not have access to other students' repositories, they have no easy way of accessing their code. This becomes a necessity when doing co-student code review, as they must somehow be given access to the relevant pull request. In this section we discuss possible solutions to overcome this problem.

### 4.1.1 Possible Solutions

A solution could be to use GitHub teams, a feature that would allow us to grant limited access to individual repositories. When a student is assigned to review someone else's pull request, such a team can be created with read rights to the relevant repository. Once the review is finished, we can either remove the team from the repository, remove the student from the team, or just delete the team altogether. This solution does however have some unwanted side effects, which can be illustrated with the following scenario.

Imagine we have two students: A and B. Student A is finished with a task from assignment 1, and therefore gets a reviewer assigned. Student B is assigned to review it, and is granted access by QuickFeed to student A's repository. Student A is also diligent, and has already started working on assignment 2. In fact, student A has nearly completed assignment 2. Student B however, has not even started on assignment 2, but because they can now access student A's assignments, they see how he/she did it.

As a consequence, using GitHub teams would mean that assignments can only be published after the deadline of the previous one is passed. A prospect that would probably not be appealing to most teachers.

Another possible solution is to create a clone repository, containing only the relevant assignment. This way, any reviewing student will only have access to the assignment in question. QuickFeed would create these repositories when needed, and then delete them once the review is complete.

Implementing this however seems highly complex, as there is a myriad of complications and problems that will have to be accounted for. For one, QuickFeed must be expanded to handle, amongst other things, the following.

- Creating clone repositories containing only the assignment that is relevant, and the accompanying pull request.

- Deleting these repositories when they are no longer needed.
- Associating actions on the original repository with the new one. I.e., we must make sure that any relevant additions to the original repository are reflected in the cloned one.
- Account for all possible complications that may occur.

An implementation like this also assumes that only one review is required per assignment, when there could in fact be several. If for example an assignment has three tasks, and therefore three related pull requests, how do we then create these clone repositories? Would we create one per pull request? This seems like a terrible approach, as the number of repositories created would become enormous in courses with many students.

Let us say that we, despite these challenges, managed to create a fully functional implementation of the above solution. We now have to cope with the prospect of having further complexified the user experience. Possibly so much so that the entire solution might be counter-productive.

### 4.1.2 Limiting Scope to Group Repositories

The two solutions proposed in the previous section either seem too complex, or somewhat suboptimal. Instead of going through with one of them, a compromise was agreed on to limit the project scope to only group assignments. This means that instead of students reviewing each other's code on regular assignments, they will only do so on group assignments. Furthermore, students will also only review other group members' pull requests.

As an example, we can imagine a group assignment with three different tasks. A group of three students would then create one pull request each, with every member solving one task. They are then each assigned, when appropriate, to review one of the other group members' pull request.

The benefit of this approach is that it avoids students needing access to other repositories all together. We only need to manage student review within a local group scope, and not for an entire course.

One disadvantage is that teachers will have to more carefully plan out the tasks they create. Say we have a group of four members, and an assignment with only three tasks. One of the tasks will then have to be shared by two of the group members. If every task is equally challenging to solve, it would put the other two members at a disadvantage. It seems then that teachers should only create as many tasks as there are designated group members, or at least a multiple of that number.

Another consideration teachers will have to account for, is that they have to create tasks that are independent of each other. If task A is dependent on task B being complete, a consequence would be group members potentially having to wait for each other.

## 4.2 Creating Pull Requests

Having limited our approach to only group assignments in the previous section, we continue by discussing how to create the pull requests themselves.

There are two viable options, both with their own set of advantages and disadvantages. Option one is to have students create the pull requests. When group assignments with tasks are created, group members can internally decide who solves which task. Option two is having QuickFeed create the individual pull requests when tasks and their respective issues are created.

### 4.2.1 Manual Creation

An advantage to letting students manually create pull requests, is that group members can self organize on who does what. If a group member thinks a certain task looks more interesting than the others, they can explicitly communicate this with the rest of the group.

Manually creating a pull request also allow us to restore a working state, in case a student incorrectly merges it. This could happen if the pull request has not been approved by a teacher. The same process we use internally to create the pull request, can be used again to recreate it.

The major disadvantage to relying on students to create pull requests, is the increased complexity that comes with it. Students will have to learn how to create pull request and when to do so, and if they mess up, they must know how to fix any potential complications. If they do not, the teaching staff must also have the capacity to assist them.

### 4.2.2 Automatic Creation

The clear advantage to having QuickFeed automatically create pull requests, is the fact that we are not imposing more complexity on students. Furthermore, if students are not involved, there is no opportunity for them to mess up. This also alleviates the teachers, as they no longer need to assist in case something goes wrong.

There is however a disadvantage to this approach. Given that QuickFeed relies on authenticating with GitHub using users specific access tokens, each pull request would have to be made either in a student or teacher's name. Both of these options seem suboptimal. Creating them as teachers seems illogical, as one teacher would then be the owner of a huge number of pull requests they do not interact with. If we create them as students, we could do so for each individual group member. This still leaves us with the issue of having to decide what group member to use when creating the pull request. If the number of issues in a repository does not align with the number of group members, what should happen? Even if we solve these problems, we are still left with the fact that individual group members can no longer self organize when solving tasks.

It seems then that any solution must create ”anonymous” pull requests, i.e., pull requests that are not directly assigned to any student. One way to do this could be to have a QuickFeed bot account. This bot account would then be the user used to create all required pull requests.

There are however drawbacks to creating anonymous pull requests as well. To facilitate student co-review, we must have a reference to the student who ”checked out” the pull request, i.e., the student actually committing to it. If we do not, we will have no way of knowing from which pool of group members to select a reviewer from. This is a problem, given the fact that a student should not be assigned to review their own pull request.

### 4.2.3 Conclusion

While option two seems ideal, there are a few key points that made us instead go with option one. First of all, the current iteration of QuickFeed does not seem to fully support it, as we have no direct way to create anonymous pull requests. Secondly, option one gives us a greater capacity to handle students incorrectly mergeing pull requests.

It is worth noting that these two approaches are not necessarily mutually exclusive, and QuickFeed could support both solutions at the same time. In fact, the initial idea was to implement them both, and thus getting all their benefits.

## 4.3 Issue Creation

Having discussed our approach on creating pull requests, we continue by exploring a similar problem. How do we create issues?

As mentioned previously, we want to create an issue on every group repository in a course, for every task in an assignment. They thereby serve as a way of actually displaying tasks to students. To create these issues, we must use the GitHub API, and again, given that GitHub authenticates on a per user basis, these issues must be created in the name of either a student or teacher.

Like with the pull request problem, creating them as a teacher seems wrong, however, creating issues as a student does not seem that problematic. It does not matter, given our context, who is assigned as an issue owner. The issues only serve as a source of information, and the only time students interact with them is when they create pull requests.

A solution could then be to simply have the first student within a group be set as the issue creator. Of course, this is a somewhat ”hacky” solution, as it makes more logical sense to have QuickFeed assigned as the owner. Which again boils down to the fundamental issue on how QuickFeed authenticates with GitHub.

## 4.4 Automated Student Feedback

To facilitate automatic feedback on individual pull requests, the initial idea was to use GitHub workflows. Mainly due to the fact that workflows are already tightly integrated into the GitHub pull request user interface.

As students work on tasks, we imagined using workflows to provide feedback based on the tests run on their code. However, as we researched ways to implement this, it became apparent that workflows were not an optimal approach. For one, to have GitHub register a workflow, a *.yml* file describing it must be present on any relevant repository. These files would have to be created and pushed to *assignments*, and then pulled by every student. The biggest problem however, is that GitHub does not support triggering workflows manually via API, and at the same time having them appear in a pull request. Meaning that we can trigger workflows remotely, but we cannot have them be displayed within a pull request on the same trigger.

As a consequence, finding an alternate solution solution became necessary. Of the options explored, the one we went with is described in the following section.

### 4.4.1 Pull Request Comments

Instead of providing automatic feedback via GitHub workflows, we can instead do so by commenting on the pull request itself. GitHub's API allows for doing this, again, with the caveat that we must comment as a student or teacher. Here, the easiest solution is to simply comment as the student that created the pull request. The comment itself can be constantly updated with new data, every time students push to their pull request.

Currently, the automatic feedback QuickFeed returns via its web interface, describes how each individual test for an assignment fared. Furthermore, it also gives a general score for the entire assignment, giving students an indication of well they did overall. With this in mind, the type of feedback returned by these comments should strive to do something similar.

# Chapter 5

# Implementation

## 5.1 Introduction

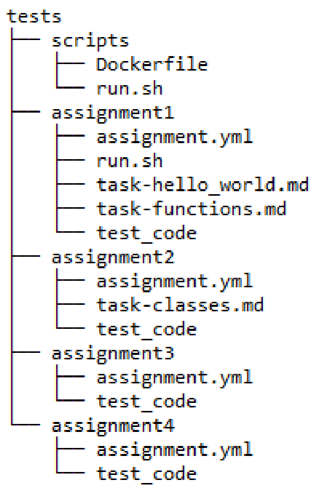
In this chapter, we explore how the project was implemented. Describing all additions that were made to QuickFeed, and some of the challenges faced.

### 5.1.1 Feature Overview

To fully grasp what to implement, this section describes all the features we want to have QuickFeed support.

#### Tasks and issues

First of all, we want QuickFeed to support teachers creating task markdown files within individual assignments in *tests*. As these are created and pushed, QuickFeed should create GitHub issues, based on their contents, in all group repositories. Furthermore, when these task files are updated or deleted, their changes should be reflected in their respective issues.



```
tests
├── scripts
│   ├── Dockerfile
│   └── run.sh
├── assignment1
│   ├── assignment.yml
│   ├── run.sh
│   ├── task-hello_world.md
│   ├── task-functions.md
│   └── test_code
├── assignment2
│   ├── assignment.yml
│   ├── task-classes.md
│   └── test_code
├── assignment3
│   ├── assignment.yml
│   └── test_code
└── assignment4
    ├── assignment.yml
    └── test_code
```

Figure 5.1: Example of a tests repository with task markdown files

### Pull Requests

For every task, group members are expected to solve them by creating a new pull request. QuickFeed should then support:

- Returning automatic feedback, in the form of a comment, every time a student pushes to a pull request.
- Automatically assigning reviewers to a pull request when students get a passing score on the task relevant for that pull request. Specifically, one teacher and one other group member should be assigned.

## 5.2 Existing implementation

An implementation has already been attempted for this project. Before starting on a new one, we first have to decide which parts of it we want to use, and which to discard. These parts are summarized in this section.

### 5.2.1 SCM Expansion

The SCM package, described in Section 2.2.6, has been expanded to support the following functions:

- CreateIssue - creates an issue on a repository.
- EditIssue - edits an issue on a repository.
- GetIssue - retrieves an issue based on issue number and repository.
- GetIssues - retrieves all issues from a repository.

CreateIssue and EditIssue proved useful, and are used with only minor adjustments to them. The two other functions, were not needed, but still used in earlier parts of the project when testing.

### 5.2.2 Logic

A lot of existing code was intended to handle the logic around managing tasks. In short, this code was meant to do determine when to create, edit, or delete issues on GitHub, based on task markdown files in *tests*. The code did however not accomplish this in a functional manner, which lead to the dilemma of whether to continue developing this faulty code, or simply start anew. In the end, we decided that any attempt to fix things would be more time consuming and confusing than simply starting fresh.

### 5.2.3 Finding and Parsing Tasks

When a teacher pushes assignments to the *tests* repository, QuickFeed will run the function UpdateFromTestsRepo. In short, this function creates a local copy of the folder structure in *tests*, and uses the *assignment.yml* files within to create assignment data objects. These are then used to update the database records for the relevant assignments. Added as a part of this process, was code to also parse all task markdown files in *tests*, and use their contents to create task data objects.

Additionally, a new field is created for the existing **Assignment** message, giving the associated data structure the capacity to hold a list of tasks. This is used to have any assignment data object created by UpdateFromTestsRepo also contain all the task data objects for that assignment.

```
repeated Task tasks = 13;  // Tasks associated with this assignment
```

Code 5.1: Modification to the Assignment message

To support parsing task markdown files, all *task-*.md* files are expected to conform to a standard format. The first line should start with the character sequence ”# <task title>”, followed by by two new line characters. Any following text there after will be treated as the task body or description.

Using this format, the function newTask creates a task data object from any task file.

```
func newTask(contents []byte, assignmentOrder uint32, name string) (*pb.Task,
    error) {
    if !bytes.HasPrefix(contents, []byte("# ")) {
        return nil, fmt.Errorf("task with name: %s, does not start with a #
            title marker", name)
    }
    bodyIndex := bytes.Index(contents, []byte("\n\n"))
    if bodyIndex == -1 {
        return nil, fmt.Errorf("failed to find task body in task: %s", name)
    }

    return &pb.Task{
        AssignmentOrder: assignmentOrder,
        Title:           string(contents[2:bodyIndex]),
        Body:            string(contents[bodyIndex+2:]),
        Name:            name,
    }, nil
}
```

Code 5.2: The newTask function

Going forward, the general approach to parsing and creating tasks is left as it is, with only minor changes to the existing code.

## 5.3 Managing Tasks and Issues

Having looked at what existing code we decided to keep, we continue by exploring the part of the project that was implemented first; how to manage tasks and issues.

As a reminder, a task is defined by the contents of a single *task-*.md* markdown file in *tests*. When referring to teachers creating, updating or deleting tasks, we mean how they create, update and delete these markdown files.

An issue on the other hand, is defined as any GitHub issue based on these tasks. For instance, we can have a course with an assignment containing three tasks. In this course, we also have five group repositories. These three tasks should then be represented on all five group repositories as GitHub issues, giving us a total of 15 issues. Tasks therefore function as a ”benchmark”, used to create issues.

### 5.3.1 Approach

To support teachers creating, updating and deleting tasks, QuickFeed must have the capacity to handle these events. For example, if a task is created, QuickFeed should create an issue based on it in every group repository within the relevant course. Similarly, as tasks are updated or deleted, these changes must also be reflected in every issue created from them. In short, we need to keep every issue in sync with its associated task.

To accomplish this, we keep a record of every task and issue in QuickFeed’s database. The task database records are used to store a reference to what a given task looked like on the latest push to *tests*. Doing so, we can at any subsequent push check if a task is in a different state than it was before, e.g., if a teacher updated its title. Furthermore, by storing a reference to every issue created, we can know which issues need to be altered. If, for example, a teacher changes the body of a task, we must update not just the existing data record for that task, but also all issues that were created from it.

The process of synchronizing both tasks and issues is handled by the new function handleTasks, which is run as part of UpdateFromTestsRepo. UpdateFromTestsRepo in turn, is run every time someone pushes to *tests*. The function handleTasks and the synchronization process in general, are further described in the following sections.

### 5.3.2 Differentiating Data Objects

When dealing with tasks and assignments, we have to differentiate how we create data objects to represent them. For instance, QuickFeed currently creates assignment data objects based on the contents of *tests*, as described in Section 5.2.3. QuickFeed also stores representations of these assignments in the database. This means that there are two types of assignment data objects we can refer to; those created from the contents of *tests*, and those retrieved from the database.

Similarly, in this project, we have to deal with the prospect of task data objects created from the contents of markdown files in *tests*, and existing ones in the database. To know which type is being referred to, we define the following abbreviations.

- TRtask - tests repository task. Meaning any task represented by a *task-*.md* markdown file in *tests*.
- DBtask - database task. Meaning any task that refers to an existing task data record stored in the database.
- TRassignment - tests repository assignment. Meaning any assignment represented by a folder and *assignment.yml* file in *tests*.

- DBassignment - database assignment. Meaning any assignment referring to an existing stored assignment data record.

### 5.3.3 Data Structures

To manage tasks and issues, two new messages are defined in the *ag.proto* file: **Task** and **Issue**.

```
message Task {
    uint64 ID              = 1;
    uint64 assignmentID    = 2;  // foreign key
    uint32 assignmentOrder = 3;
    string title           = 4;
    string body            = 5;
    string name            = 6;
    repeated Issue issues  = 7;  // Issues that use this task as a benchmark
}
```

Code 5.3: Task message

The title and body fields are defined by the contents of a task markdown file. These fields define the title and body of all issues that are created from this task.

For an assignment, order is defined as a number, used to determine the order in which it is represented in a course. It is set by a teacher in the *assignment.yml* file, as described in Section 2.2.2. By allowing QuickFeed to associate TRassignments with DBassignments, it functions as a local assignment reference within a single course. Similarly, when creating TRtasks we store a reference to the TRassignment it was found in, by using the TRassignment's order. If our scope is limited to a single course, this allows us to know which DBassignment a given TRtask is supposed to be a part of.

The name field is used to associate TRtasks with DBtasks. If a task markdown file with the name *task-hello_world.md* is found within *assignment1*, then its corresponding name will be assignment1/hello_world. This name is set when the task itself is parsed, as described in Section 5.2.3.

```
message Issue {
    uint64 ID           = 1;
    uint64 repositoryID = 2;  // Represents the internal ID of a repository
    uint64 taskID       = 3;  // Task that this issue draws its content from
    uint64 issueNumber  = 4;  // Issue number on scm. Needed for associating db
        issue with scm issue
}
```

Code 5.4: Issue message

When compiled, the generated data structure is used to represent issues internally in QuickFeed. The issueNumber field lets us reference any GitHub issue within a repository.

### 5.3.4 Synchronizing Tasks

As mentioned, we store references to tasks in the database. When teachers push new or updated tasks to *tests*, QuickFeed must create or update the existing DBtask records representing them. Similarly, when a task has been deleted, its associated DBtask must also be deleted from the database.

These events are handled by the database method SynchronizeAssignmentTasks, which is run at the start of handleTasks. SynchronizeAssignmentTasks is supplied with every TRassignment created by UpdateFromTestsRepo, each with their list of TRtasks. They are then used to create the following mapping.

```
taskMap[assignmentOrder][taskName] = TRtask
```

Code 5.5: Task mapping

This mapping allows us to loop through all DBassignment records in a course, and associate each of its DBtasks with a TRtask.

The process of synchronizing tasks is explained as follows. For every DBtask in a DBassignment, check if it has the same name as any of the TRtasks belonging to the TRassignment with the same order. If no match can be found for the DBtask, it must mean that the TRtask has been removed, and the task itself no longer exists, which means that the DBtask can be safely deleted. However, if there is a match for the DBtask, we check if its body or title differs from its TRtask counterpart. If true, we conclude that the task has been updated, and if not, it must be unchanged. Furthermore, any TRtask that is not represented by a DBtask must represent a new task, and we therefore create a DBtask to represent it.

The main synchronization logic performed by SynchronizeAssignmentTasks is listed in Code 5.6. Note that error handling has been removed for simplicity.

```go
for _, DBassignment := range DBassignments {
    var DBtasks []*pb.Task
    tx.Find(&DBtasks, &pb.Task{AssignmentID: DBassignment.GetID()})
    for _, DBtask := range DBtasks {
        TRtask, ok := taskMap[DBassignment.Order][DBtask.Name]
        if !ok {
            // Existing task in database not among the supplied tasks
            tx.Delete(DBtask).Error
            DBtask.MarkDeleted()
            updatedTasks = append(updatedTasks, DBtask)

            // Find issues associated with the existing task and delete them
            var issues []*pb.Issue
            tx.Delete(issues, &pb.Issue{TaskID: DBtask.ID})
            continue
        }
        if DBtask.HasChanged(TRtask) {
            // Task has been changed and must be updated
            DBtask.Title = TRtask.Title
            DBtask.Body = TRtask.Body
            updatedTasks = append(updatedTasks, DBtask)
            tx.Model(&pb.Task{}).
                Where(&pb.Task{ID: DBtask.GetID()}).
                Updates(DBtask).Error
        }
        delete(taskMap[DBassignment.Order], DBtask.Name)
    }

    // Find new tasks to be created for the current assignment
    for _, TRtask := range taskMap[DBassignment.Order] {
        TRtask.AssignmentID = DBassignment.ID
        createdTasks = append(createdTasks, TRtask)
    }
}
```

Code 5.6: Task synchronization performed by SynchronizeAssignmentTasks

SynchronizeAssignmentTasks also returns the DBtasks it has created or updated, which are used when synchronizing issues.

### 5.3.5 Synchronizing Issues

Having synchronized tasks, we must also create, update and delete all GitHub issues where necessary, as well as their relevant database records.

As mentioned, SynchronizeAssignmentTasks already determines the tasks that have been created, updated or deleted. In fact, when it detects that a task has been deleted, it deletes not only the task but all associated issue records, as seen on Line 14 in Code 5.6. Issue data records also never need to be updated, since they carry no other non-relational information than an issue number, which always remains static. The only remaining database related synchronization for issues is therefore creating them.

The entire process of synchronizing issues happens in handleTasks, and is shown in Code 5.7.

```go
createdIssues := []*pb.Issue{}
for _, repo := range repos {
    if !repo.IsGroupRepo() {
        continue
    }
    repoCreatedIssues, err := createIssues(ctx, sc, course, repo, createdTasks)
    if err != nil {
        return err
    }
    createdIssues = append(createdIssues, repoCreatedIssues...)
    if err = updateIssues(ctx, sc, course, repo, updatedTasks); err != nil {
        return err
    }
}
// Create issues in the database based on issues created on the scm.
return db.CreateIssues(createdIssues)
```

Code 5.7: Issue synchronization performed by handleTasks

In short, we use the created and updated DBtasks returned by SynchronizeAssignmentTasks, as arguments in createIssues and updateIssues respectively. These functions use the SCM functions described in Section 5.2.1. Then, by looping through every course group repository, GitHub issues are created and updated accordingly. The function createIssues also returns a list of all the issues it created. These are used at the end of the process to create issue database records.

It is worth mentioning that there is no description of how we delete GitHub issues in the above process. To delete issues, we wanted to expand the SCM API to give it that capacity. It turns out however, that GitHub's REST API does not support deleting issues. An alternative could have been to use GraphQL, a query language for API's, but instead we went for a simpler solution. In place of deleting issues, they are closed, and their title and body are inserted with a statement indicating that the associated task has been deleted.

## 5.4 Managing Pull Requests

Having implemented support for tasks and issues, we must now do the same for pull requests.

### 5.4.1 Pull Request Data Structure

To manage pull requests we need a data structure to represent them. For this purpose, the **PullRequest** message is created.

```
message PullRequest {
    enum Stage {
        NONE     = 0;
        DRAFT    = 1;
        REVIEW   = 2;
        APPROVED = 3;
    }
    uint64 ID                   = 1;
    uint64 externalRepositoryID = 2; // Represents the external repository ID
    uint64 taskID               = 3; // Foreign key
    uint64 issueID              = 4; // Foreign key
    uint64 userID               = 5; // The user who owns this PR
    uint64 commentID            = 6; // GitHub ID of the comment used for
        automatic feedback
    string sourceBranch         = 7; // The source branch for this pull request
    uint64 number               = 8; // Pull request number
    Stage stage                 = 9;
}
```

Code 5.8: PullRequest message

It holds a reference to the issue that it was created for, as well as the task that serves as the benchmark for said issue. These references allow us to quickly relate any pull request to its corresponding task and issue.

The stage field is used to keep track of the ”stage” a pull request is in. When a pull request is first created, it is in the draft stage. Once enough of its tests are passing, it moves to the review stage, meaning that it is now ready for review. The final stage, approved, is reached when a teacher actually approves the pull request. How these stages are traversed, and how we know when to go from one to another, is described in the following sections.

### 5.4.2 Creating Pull Requests

As students create pull requests to solve issues/tasks, we must make sure that a pull request record is created and stored internally to represent them. Doing so allows us to keep track of which student is solving what task, who should be assigned to review the pull request, and more. This section describes how this is implemented.

In Section 4.2 we decided to have students manually create pull requests. To accommodate this feature, QuickFeed must react to the pull requests being opened. Expanding QuickFeed to also handle pull request related webhooks, allows us to do just that.

Specifically, when a pull request is opened, the handlePullRequestOpened function is run. It will check if a pull request is legitimate, i.e., it was created on a group repository and is linked to a valid issue. If successful, the function creates a new data object to represent the pull request, by using the data in the event payload, and then stores it in the database.

Associating a pull request to an issue requires the students themselves explicitly doing so. GitHub allows you to link issues to pull requests on their creation, by inserting a reference to them in the pull request body. Doing so, we thought that the linked issue would also be part of the webhook payload, however, this turned out to be incorrect. Instead we have to parse the issue number from the pull request body itself, as shown below.

```go
func getLinkedIssue(body string) (uint64, error) {
    if count := strings.Count(body, "#"); count != 1 {
        return 0, errors.New("pull request body does not contain exactly one '#'
             character")
    }
    subStrings := strings.Split(body, "#")
    issueNumber, err := strconv.Atoi(subStrings[len(subStrings)-1])
    if err != nil {
        return 0, fmt.Errorf("failed to parse issue number from pull request
            body: %w", err)
    }
    return uint64(issueNumber), nil
}
```

Code 5.9: The getLinkedIssue function

This solution somewhat limits the possible contents of a pull request body. For example, students must make sure only to use one ”#” character for the function to succeed. Possible ways to improve this approach is further discussed in section 6.3.3.

### 5.4.3 Closing Pull Requests

When students close and merge pull requests—ideally when they are supposed to—QuickFeed must also act accordingly. Again, we use webhooks to listen to pull request closed events, and single out those that are relevant to QuickFeed. Relevant in this context, is any event for a pull request that already exist in QuickFeed’s database, i.e., they were created as part of the process described in the previous section. Events for pull requests that are closed, but not merged, can also be filtered out, as these constitute an incorrect close. If students do happen to close pull requests like this, a working state can be restored by the student, simply by reopening it. QuickFeed doing nothing in these cases is therefore the simplest solution.

Assuming that the event is valid, there are two possible outcomes. Either the pull request is approved, and it and its associated issue can safely be deleted from the database. Or, it has been closed and merged by a student without being explicitly approved by a teacher. If this happens, QuickFeed must have the capacity to restore a working state for the issue/task in question. To handle these situations, QuickFeed deletes only the pull request and not the issue record. This way, a student doing an incorrect merge can simply reopen the issue that was closed, and then create a new pull request to represent it. Thereby restoring a working state with little QuickFeed involvement.

## 5.5 Scoring Tasks

To allow for both automatic and manual feedback, we need a way to score tasks. Delivering automatic feedback requires us to do so solely on the tests associated with any given task. Manual feedback on the other hand, needs some sort of task related score to determine when to assign reviewers. Currently, QuickFeed only tests and scores student code based on an assignment as a whole. Having it do so on a per task basis therefore seems like a necessity to fulfill our goals.

QuickFeed's ci package is fundamentally designed to test assignments, therefore rewriting it to support tasks would be challenging. Any solution might also not be backwards compatible. To implement our features though, QuickFeed does not need to explicitly test task related code, only score it. As such, we can instead complement the score package to support this functionality.

Doing so turned out to be pretty straight forward. First, the score message described in Section 2.2.3 is modified to also include a task name field. Doing so, gives us a direct reference to the task a given score is associated with. Furthermore, to support teachers adding task specific tests, two new variants of the existing Add and AddSub methods are created. These allow teachers to specify the task name a test should be a part of. Finally, to generate a task specific total score, QuickFeed simply sums over the scores that have that task's name.

It should be noted that when referring to a task name in the context of the score package, we mean the local task name. The local task name differs from the regular one by omitting the assignment name. As an example, a task with the name assignment1/hello_world will have the local name hello_world. This was primarily done to make the process of associating tests and tasks more intuitive for the teachers doing so.

Given that students will be working on non-default local branches when solving tasks, we also altered the ci package to support checking out these branches. As mentioned in Section 2.2.4, QuickFeed supports supplying the run script with arguments. By providing the script with the branch name, we can thereby switch to the correct local branch before running tests.

```
# Checkout branch if it is not main
if [ {{ .BranchName }} != "main" ]; then
  git checkout {{ .BranchName }}
fi
```

Code 5.10: Checking out non-default local branch

With these changes implemented, we can generate a score from 0 - 100 for specific tasks, just as is already possible for entire assignments.

## 5.6 Manual Feedback

Having implemented QuickFeed support for scoring tasks in the previous section, these features can now be used to facilitate manual review. In short, our approach is to assign one student and teacher reviewer to a pull request when a passing score is achieved. This passing score limit is the same as a given task's assignment score limit.

Checking whether a task is passed or not must be done every time a student pushes to a group repository. QuickFeed already has a function that handles these push events, and we can therefore manage all review assignment related logic there.

### 5.6.1 Assigning Reviewers

To handle pushes to non-default branches in group repositories we implemented the handlePullRequestPush function. Its primary job is to attempt to find a pull request that is associated with a given branch push. In the event payload, there is information about both the branch and the repository that was pushed to. Using these, QuickFeed queries the database for the specific pull request that is associated with them.

If a pull request exists for the repository and remote branch, it is used to retrieve the task associated with it. Doing so allows us to get the specific task name, which can then be used to retrieve the task specific total score. Finally, if this score is greater than the score limit, the assignment process can begin.

Before assigning reviewers there are a few things that have to be accounted for. First of, we need to make sure not to assign the same reviewers over and over. To illustrate, we do not want one teacher to review every pull request in a course, with the rest of the teachers not being assigned to review anything. Similarly, having one group member review every pull request within a group is also unwanted. Secondly, the owner of a pull request must not be assigned as its reviewer.

To tackle the first problem, we use two maps, one for teachers and one for group members, to facilitate in-memory storage of every reviewer. Specifically, these maps track the total amount of times every reviewer has been assigned to a pull request. They are described as follows, and are as shown, in fact maps of maps.

```
teacherReviewCounter[courseID][userID]  = count
groupReviewCounter[groupID][userID]     = count
```

Code 5.11: Maps used to store review counts

These are then used as arguments in the function getNextReviewer. getNextReviewer's purpose is to find, amongst the supplied users, the one that has the least amount of total reviews, and then return said user. By supplying it with all course teachers or a group's members, it finds the next eligible teacher or student reviewer respectively. To tackle the second problem mentioned earlier, we make sure to to not include the pull request owner in the user list.

```go
func getNextReviewer(ID uint64, users []*pb.User, reviewCounter map[uint64]map[
    uint64]int) (*pb.User, error) {
    if len(users) == 0 {
        return nil, errors.New("list of users is empty")
    }
    reviewerMap, ok := reviewCounter[ID]
    if !ok {
        // If a map does not exist for a given ID we create it,
        // and return the first user.
        reviewCounter[ID] = make(map[uint64]int)
        reviewCounter[ID][users[0].GetID()] = 1
        return users[0], nil
    }
    userWithLowestCount := users[0]
    lowestCount := reviewerMap[users[0].GetID()]
    for _, user := range users {
        count, ok := reviewerMap[user.GetID()]
        if !ok {
            // If the user is not present in the review map,
            // then they are returned as the next reviewer.
            reviewerMap[user.GetID()] = 1
            return user, nil
        }
        if count < lowestCount {
            userWithLowestCount = user
            lowestCount = count
        }
    }
    reviewerMap[userWithLowestCount.GetID()]++
    return userWithLowestCount, nil
}
```

Code 5.12: The getNextReviewer function

In order to actually assign reviewers to a GitHub pull request, the SCM API is expanded with the function RequestReviewers. It is supplied with the GitHub login usernames of the retrieved reviewers. If successful, the entire process is finished by updating the pull request stage to review, signaling that it is now being reviewed.

### 5.6.2 Approving Pull Requests

Having described how reviewers are assigned to pull requests in the previous section, this section explores what happens when teachers approve them.

The function handlePullRequestReview handles everything related to pull request reviews. It is triggered by pull request review related webhooks, and uses the accompanying payload to find the relevant pull request record. If one exists, it attempts to find the user that posted the review. The reason why the specific reviewer is needed, is because we must make sure that the review is from an actual teacher. Otherwise, the group member that was assigned to review could also internally approve the pull request. So, as long as the review is from a teacher, the pull request stage is updated to approved, meaning that it can be safely merged.

## 5.7 Automatic Feedback

Using QuickFeed's new capacity to score tasks, we implemented support for manual feedback on pull requests. This new capacity also allows for automatic feedback on the same pull requests.

Our approach is to use a single comment on a student's pull request to automatically publish test results from their latest commit. This comment must be updated every time the student pushes to their pull request, and as such we can therefore use the function handlePullRequestPush. A function that is used to manage all non-default branch pushes to group repositories, as described in Section 5.6.1.

### 5.7.1 Formatting Feedback Comments

To create the feedback comment itself, we must format one based on the test results for a given push. For this purpose, we create the function CreateFeedbackComment.

The general idea is to use the task specific scores, as part of the test results for an assignment, to generate a table. For every task specific test, we can describe its received score, the weight of the test, and how much the results from that test counts towards the total score. GitHub lets you create tables in comments by using a special string formatting. With this in mind, CreateFeedbackComment is implemented as described in Code 5.13.

```go
func CreateFeedbackComment(results *score.Results, task *pb.Task, assignment *pb
    .Assignment) string {
    body := "## Test results from latest push\n"
    body += "| Test Name | Score | Weight | % of Total |\n"
    body += "| :-------- | :---- | :----- | ---------: |\n"

    for _, score := range results.Scores {
        if score.TaskName != task.LocalName() {
            continue
        }
        percentageScore := (float64(score.Score) / float64(score.MaxScore)) * (
            float64(score.Weight) / results.TotalTaskWeight(task.LocalName()))
        body += fmt.Sprintf("| %s | %d/%d | %d | %.2f%% |\n", score.TestName,
            score.Score, score.MaxScore, score.Weight, percentageScore*100)
    }
    body += fmt.Sprintf("| **Total** | | | **%d%%** |\n\n", results.TaskSum(task
        .LocalName()))
    body += fmt.Sprintf("Once a total score of %d%% is reached, reviewers are
        automatically assigned.\n", assignment.GetScoreLimit())
    return body
}
```

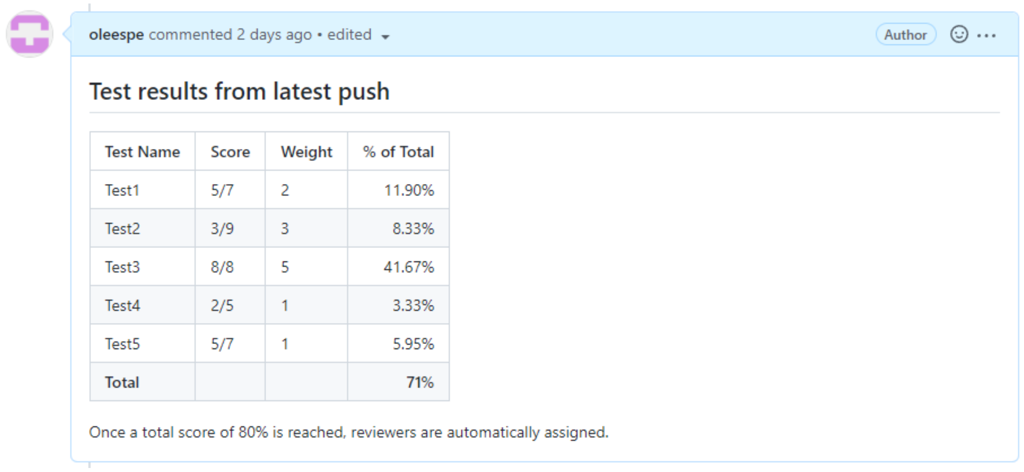
Code 5.13: The CreateFeedbackComment function

To illustrate, we can supply CreateFeedbackComment with the results, task and assignment described in Code 5.14.

```
results := &score.Results{
    Scores: []*score.Score{
        {TestName: "Test1", TaskName: "1", Score: 5, MaxScore: 7, Weight: 2},
        {TestName: "Test2", TaskName: "1", Score: 3, MaxScore: 9, Weight: 3},
        {TestName: "Test3", TaskName: "1", Score: 8, MaxScore: 8, Weight: 5},
        {TestName: "Test4", TaskName: "1", Score: 2, MaxScore: 5, Weight: 1},
        {TestName: "Test5", TaskName: "1", Score: 5, MaxScore: 7, Weight: 1},
        {TestName: "Test6", TaskName: "2", Score: 5, MaxScore: 7, Weight: 1},
        {TestName: "Test7", TaskName: "3", Score: 5, MaxScore: 7, Weight: 1},
    },
}
CreateFeedbackComment(results, &pb.Task{Name: "lab1/1"},
                        &pb.Assignment{Name: "lab1", ScoreLimit: 80})
```

Code 5.14: Example of a CreateFeedbackComment run

The resulting comment body in Figure 5.2 is then generated. Note that the tests with name Test6 and Test7 are omitted from the table, because they are not associated with the task supplied to CreateFeedbackComment.



Test results from latest push

| Test Name | Score | Weight | % of Total |
|---|---|---|---|
| Test1 | 5/7 | 2 | 11.90% |
| Test2 | 3/9 | 3 | 8.33% |
| Test3 | 8/8 | 5 | 41.67% |
| Test4 | 2/5 | 1 | 3.33% |
| Test5 | 5/7 | 1 | 5.95% |
| Total | | | 71% |

Once a total score of 80% is reached, reviewers are automatically assigned.

Figure 5.2: Feedback comment on a pull request

### 5.7.2 Publishing Feedback Comments

To actually publish the feedback comments to a GitHub pull request, we again expand the SCM API. The function CreateIssueComment is used to create the comments, while EditIssueComment updates existing ones. The reason for using

”Issue” in the function names instead of ”PullRequest”, is because GitHub treats pull requests as issues with code. GitHub API calls to comment on issues and pull requests are therefore the same.

The first time a student pushes to a pull request, we use CreateIssueComment to create the initial comment. CreateIssueComment returns the ID of the comment that was created, which is then stored as a field on the relevant pull request record. Any subsequent pushes to the same pull request will then instead use EditIssueComment, which takes that ID as an argument. Doing so, only one comment is present on the student’s pull request, and is constantly updated with new relevant data.

# Chapter 6

# Insights

In this chapter we describe insights regarding this projects implementation.

## 6.1 User Experience

The features implemented in this project impose new demands on both students and teachers wanting to use them. The most apparent of them are listed in this section.

Students solving group assignments with tasks, must be able to, or made aware of the following:

- How GitHub issues work, and the fact that they are representations of tasks that teachers have created.
- How to create new branches and commit to them.
- How to create pull requests from these branches, and link the correct issue to them.

- How to actually solve tasks. Meaning that they must know which parts of an assignment is relevant for a given task.
- How the general flow of an assignment with tasks goes. In essence, they must know the three stages that are relevant for pull requests; draft, review and approved.
- How to close and merge pull requests when they are approved.
- How to handle situations where something goes wrong. For example, if they create a pull request without correctly linking the relevant issue.

Similarly, teachers and the teaching staff in general must know the following:

- How to create tasks via markdown files.
- How to associate tests with tasks, using the score package.
- How to help students failing with any of the points in the previous list.

Especially for students, there is a lot of new ”stuff” to deal with. Any assignment would have to carefully explain the points mentioned above.

## 6.2 Known Limitations

There are certain known limitations to this project’s implementation. These limitations were discovered throughout the project, but no attempt to fix them was made, mainly due to the fact they were considered edge cases. They are listed in this section.

### 6.2.1 Differing Branch Names

There is a weakness related to students potentially having remote and local branches with different names.

To facilitate testing and scoring of student code on non-default local branches, we have to check out that local branch. We get the branch name to check out, through the pull request opened event payload. This branch name however, is for the remote GitHub repository the pull request was created for. Students can in theory create a local branch called ”branch1”, and push it to the remote branch ”branch2”. If this is done, QuickFeed has no reference to the local branch it needs to run tests on.

### 6.2.2 Human Error

There are certain actions students can perform, that QuickFeed simply cannot handle with the current implementation.

For instance, if students were to manually delete the issues that QuickFeed creates, there would be no way of recreating them. This would break the entire assignment process, as students can no longer create pull requests for these issues. Similarly, students can also delete the feedback comments created on their pull requests. If this happens, an error will occur when QuickFeed tries to publish new feedback to the deleted comment.

Of course, these are fringe cases, as they require students doing something they would probably know to be incorrect.

### 6.2.3 Manually Graded Assignments

Post implementation, a problem was encountered with how QuickFeed treats manually graded assignments.

To QuickFeed, a manually graded assignment is any assignment whose reviewers field is greater than zero. When handling push events for such assignments, no automated tests are run. No automated tests, means no test scores and no assignment total score.

In our implementation, we use scores for two purposes. One is to automatically publish feedback on a pull request, while the other is to determine when to assign reviewers to a pull request. If an assignment generates no scores, we have no way of performing either of these tasks. It seems then that any assignment taking advantage of the features developed in this project cannot be manually graded.

## 6.3 Possible Improvements

Certain parts of the implementation has the potential to be improved. These improvements primarily revolve around already implemented code, and are relatively minor in scale. They are therefore discussed here, while any larger scale project is instead proposed in Chapter 7.

### 6.3.1 GitHub App

When discussing how to create issues and pull requests in sections 4.3 and 4.2 respectively, we explored the need for QuickFeed to be able to identify as ”itself”. Implementing QuickFeed as a GitHub App would accomplish this, as a GitHub App acts on its own behalf when interacting with the GitHub API [5].

Running concurrently with this project, was another that converted QuickFeed into a GitHub App. For this implementation to be fully optimal, it should take advantage of this fact. For instance, creating issues should be done as QuickFeed, and not as students. Similarly, while creating pull request feedback comments as students is not that problematic, it would be best if they too were created as QuickFeed.

QuickFeed being a GitHub App, also opens the door for pull requests being created automatically, as discussed in Section 4.2. Any implementation would still have to solve the problem of deciding the owner of a given pull request. To clarify, even if pull requests are created anonymously, QuickFeed still needs to know which student is actually working on it. Otherwise, we have no way of knowing from which pool of group members we should assign reviewers, as explained in Section 5.6.1.

### 6.3.2 Task Naming Format

A problem with the current implementation, is the fact that there are two types of task names.

As discussed in Section 5.3.3, the task data structure has a field name. It is set upon its creation, and is a combination of the assignment it is for, and the name of the markdown file that describes it. For example, a task can have the name assignment1/hello_world. When complementing the score package to also support tasks, we instead use a local task name, i.e., only hello_world.

Originally, including the assignment name was necessary to associate tasks with each other when synchronizing them. Since then, the implementation has changed, and including the assignment name is no longer needed. A recommendation is therefore to make the local task name standard, thereby avoiding any unwanted confusion.

### 6.3.3 Linking Issues

When students create pull requests, a requirement is that they also link the relevant issue, by inserting a reference to them in the pull request body. The implementation for this has two problems. First of, the function that parses the issue from the pull request body, described in Code 5.9, is limited in its capabilities, and demands a certain format. Secondly, if students were to fail, when linking pull requests to issues, they are not immediately made aware.

To link an issue to a pull request, GitHub supports using a keyword such as ”Fixes” or ”Closes”, followed by the character sequence ”#<issue number>”. As an example, linking issue #30 to a new pull request can be done by inserting ”Fixes #30” in the pull request body. The function that parses the issue number from this sequence is limited to only handling simple cases, as it only splits the pull request body. A better solution would probably be to use a regular expression search.

To tackle the second problem, a solution could be to insert a comment or something similar in the faulty pull request, stating that an associated issue could not be found. Another approach could be to support linking issues after the pull request has already been created. Thereby removing the need for students to create a completely new pull request.

### 6.3.4 Support for Regular Assignments

While our implementation is intended primarily for group assignments, expanding most parts of it to support regular assignments should not be difficult. In fact, the only feature we have implemented that works explicitly on group assignments, is the co-student review portion. As discussed in 4.1, the reason we limited the project’s scope to only group assignments, was because of the problem of how to give students access other student repositories. All other features, such as creating issues from tasks, and managing pull request pushes, should be implementable for regular assignments.

# Chapter 7

# Future Work

Much of what is implemented serves only as a baseline, compared to the vast range of possible future projects. This chapter discusses some possible projects that expand on this one.

## 7.1 Further QuickFeed Integration

Even though QuickFeed now supports pull requests, issues and tasks, they are not highly integrated with the rest of QuickFeed. The features developed in this project are mostly standalone, and not generally connected with QuickFeed's existing ones.

To illustrate, we can imagine a teacher creating an assignment with several tasks. A group could in theory correctly solve all parts of this assignment, push it to their group repository, and then have it scored the regular way. They would still get a passing score in QuickFeed's web interface, all the while having skipped the entire pull request process. Similarly, doing things the intended way also has no direct consequence on how QuickFeed treats the assignment submission as a whole. Having a pull request approved, has no impact on how an assignment is approved. Only when a teacher approves the assignment via the web interface, is it actually approved.

A suggestion is therefore to further integrate the implemented features with QuickFeed's assignment submission process. For example, a requirement to getting an assignment approved, could be to have all relevant pull requests approved. To support this, one would have to develop a system that compares the number of approved pull requests, to the number of tasks for an assignment. Implementing this would also require a rethink of the entire assignment approval process.

If such a system is developed, there is also the potential to further integrate it with QuickFeed's existing web interface. For example, assignments that rely on pull request approval, can be made to not require explicit approval via the web interface. The assignment can instead be automatically approved once all relevant pull requests are. This way, the need for a custom web interface diminishes.

Another way to move away from a custom web interface to simply relying on GitHub's, could be to expand on the feedback given in the pull request comments. Technically, all data that is used for the current automatic feedback system, is also available to be published in these comments. Of course, there might be limitations to the amount of information a pull request comment can reliably express, but the potential is still there.

## 7.2 Further GitHub Integration

There is also the potential to further integrate with GitHub.

For instance, GitHub's Checks API seems to have a lot of potential [7]. In short, it can be used to create custom checks on a pull request, by sending special webhook events every time someone pushes code to a repository. These checks can then be used on a pull request to signals whether it has been correctly approved or not. Furthermore, if we only allow pull request merges when this check passes, we can avoid students incorrectly merging pull requests altogether. GitHub supports limiting access like this by specifying rules on repository branches [6].

# Chapter 8

# Conclusions

This thesis had the goal of using GitHub's issue and pull request features to further expand on how students receive feedback when working on assignments. By implementing QuickFeed support for teachers subdividing assignments into tasks, we have created a new way for teachers to approach making assignments. Furthermore, by expanding QuickFeed's existing score package, we now support testing and scoring tasks. This allows us to provide automated feedback directly on pull requests, as well as manual feedback by teachers and co-students on student code.

We believe that both students and teachers can benefit from the features implemented in this project. Giving students an avenue for direct feedback on their code through pull request code reviews, seems especially beneficial. Furthermore, familiarising students with GitHub pull requests can be highly beneficial for them in the long run.

Even though these features are functional, there is still much that can be done to improve them. In this sense, what has been accomplished in this project, should hopefully serve as a good baseline for future projects.

# Bibliography

[1] Atlassian. *Webhooks*. URL: `https://developer.atlassian.com/server/jira/platform/webhooks`. (accessed: 09.04.2022).

[2] A.B. Brynildsen, J.H. Lindhom, and M. Bjerga. “Quickfeed: Redesigned Web Frontend”. Bachelor’s Thesis. University of Stavanger, 2021.

[3] T. Darvik and T. Gliniecki. “Building Autograder front-end with ReactJS”. Bachelor’s Thesis. University of Stavanger, 2016.

[4] H. Furubotten. “The autograder project: Improving software engineering skills through automated feedback on programming exercises”. MA thesis. University of Stavanger, 2015.

[5] GitHub. *About Apps*. URL: `https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps`. (accessed: 07.05.2022).

[6] GitHub. *About protected branches*. URL: `https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches`. (accessed: 08.05.2022).

[7] GitHub. *Checks*. URL: `https://docs.github.com/en/rest/checks`. (accessed: 08.05.2022).

[8] Google. *Protocol Buffers*. URL: `https://developers.google.com/protocol-buffers`. (accessed: 12.05.2022).

[9] A. Khurshid. *Quickfeed Support for Feedback via Pull Requests and Issues*. University of Stavanger, 2021.

[10] D. Urdal and V. Yaseneva. “Experience Report: Replacing REST with gRPC in Autograder”. Bachelor’s Thesis. University of Stavanger, 2019.