Teaching and Improving Code Review in the Classroom

by

Mary Z Zhong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Robert C. Miller
Distinguished Professor of Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Teaching and Improving Code Review in the Classroom

by

Mary Z Zhong

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Code review in the classroom setting presents novel challenges compared to code review in the software engineering industry. In particular, students are not as experienced as professional software engineers. 6.031: Elements of Software Construction is an undergraduate computer science class at MIT that serves as an introduction to software engineering. In this class, Caesar, an online code review tool, is used for peer review of student code. Our work uses crowdworking research and principles to make Caesar even more suitable for the classroom setting. First, we provide more structure to writing code review comments. Next, we provide an interface for staff to evaluate the quality of student comments. Finally, we introduce practice tasks, which act as an initial training period at the beginning of every code review session. In this way, we give students more knowledge and structure to write high quality code review comments, and we provide an interface to evaluate that quality. We believe that our work allows students to have a richer learning experience through Caesar.

Thesis Supervisor: Robert C. Miller
Title: Distinguished Professor of Computer Science

Acknowledgments

Thank you to my thesis advisor, Rob Miller, for providing guidance during the entire course of this work. He was able to keep me on track during a trying semester and was always available to work through tough problems and come up with creative solutions. Thank you to my fellow research group members as well—UP standUPs were very fun.

A special thank you to 6.031 teaching assistants and lab assistants, as well as the instructors, Rob Miller and Max Goldman. They really made my three semesters of being a TA, along with my thesis work for 6.031, worth it. They were available to provide testing and immensely helpful feedback of my work on Caesar. Another thank you goes to Max for being my academic advisor during my time as a computer science major at MIT. Thank you for being supportive, responsive, and always available for a chat.

Next, thank you to the people around me during this MEng year at MIT. Thank you to Julia, who supported me as a friend, research group mate, and classmate throughout my undergrad and MEng. Thank you to Grace for keeping me sane while writing this thesis.

Thank you to the previous students whose work and research have made Caesar what it is today.

Finally, thank you to my family, who has supported and encouraged me throughout the years. The completion of this work and my accomplishments at MIT is in large part due to their sacrifices and support.

Contents

1 Introduction **13**

2 Related Work **15**

2.1 Caesar and 6.031 15

2.2 Feedback and Crowdsourcing 16

2.3 Crowdsourcing Quality 17

3 Code Review Interface and Process **19**

3.1 Structured Comments 19

3.2 Grading Student Code Review 20

3.3 Practice Tasks 21

3.3.1 Starting Code Review 21

3.3.2 Practice Task Workflow Overview 22

4 Practice Task Implementation **27**

4.1 Backend Models 27

4.1.1 `PracticeProblem` 28

4.1.2 `ExpectedAnswer` 29

4.1.3 `PracticeTask` 29

4.2 Practice Task Set Up 30

4.2.1 Practice Task Assignment 33

4.3 Practice Task Flow and Automated Feedback 34

4.4 Fall 2019 Semester 37

4.4.1 Assigning Practice Tasks . . . 37
4.4.2 Generating Automated Comments . . . 37

5 Evaluation **41**
5.1 Spring 2020 Semester . . . 41
5.2 Deployment in 6.031 . . . 42
5.3 Code Review Participation . . . 43
5.3.1 Quantity of Comments . . . 43
5.3.2 Quality of Comments . . . 47
5.4 Practice Task Accuracy . . . 50
5.4.1 Fall 2019 . . . 51
5.4.2 Spring 2020 . . . 54
5.5 Additional Practice Task Experiments . . . 58
5.5.1 Effectiveness of Keywords . . . 58
5.5.2 Spring 2020 Problem Set 3 Experiment . . . 61

6 Discussion **63**
6.1 `sfb/etu/rfc` Tags . . . 63
6.2 Practice Task Feedback Accuracy . . . 63
6.3 Effect of Practice Tasks on Regular Code Review Tasks . . . 64

7 Future Work **67**
7.1 Keywords Accuracy . . . 67
7.2 Keyword Robustness . . . 68

8 Conclusion **69**

List of Figures

3-1 Caesar's new comment UI 19
3-2 A written comment with three tags 20
3-3 A student comment from the LA view 21
3-4 "Start Reviewing" button in Caesar 22
3-5 Old Caesar interface - list of tasks 22
3-6 "Continue Reviewing" button in Caesar 22
3-7 Practice task: instructions and initial interface 23
3-8 Practice task: student makes a comment 24
3-9 Practice task: automated feedback 25

4-1 Automated feedback: at least one expected keyword 36
4-2 Automated feedback: no expected keywords 36
4-3 Automated feedback: missing comment 36
4-4 Practice task file from Fall 2019 38

5-1 Characters per comment - sp19, fa19, sp20 44
5-2 Comments per student per task - sp19, fa19, sp20 45
5-3 Plus graded comments 48
5-4 Check graded comments 48
5-5 Minus graded comments 49
5-6 ps3 practice task accuracy - fa19 52
5-7 ps4 practice task accuracy - fa19 54
5-8 ps0 practice task accuracy - sp20 55
5-9 ps1 practice task accuracy - sp20 56

5-10 ps2 practice task accuracy - sp20 56
5-11 ps3 practice task accuracy - sp20 57
5-12 ps4 practice task accuracy - sp20 58
5-13 ps0 keyword usage - sp20 59
5-14 ps1 keyword usage - sp20 59
5-15 ps2 keyword usage - sp20 60
5-16 ps3 keyword usage - sp20 60
5-17 ps4 keyword usage - sp20 60

List of Tables

5.1 Comment quantity: Spring 2019 semester 43
5.2 Comment quantity: Fall 2019 semester 44
5.3 Comment quantity: Spring 2020 semester 44
5.4 LA comment grading: Fall 2019 semester 47
5.5 LA comment grading: Spring 2020 semester 48

Chapter 1

Introduction

Code review is used widely in industry as part of the software development process, and it provides a way for coworkers to give each other feedback on code, as well as gain knowledge about the kinds of projects team members are working on. Code review also has potential in being used in a classroom setting. In particular, 6.031: Elements of Software Construction, an undergraduate computer science course at MIT, utilizes Caesar, an online code review tool created for the class. Caesar allows students to review other students' problem set code, along with staff feedback and moderation.

When thinking about code review in the classroom, there are differences between industry and the classroom setting that must be considered. First, 6.031 is an introduction to software engineering, and students will not have had prior experience with code review. Because of this, we want to be able to teach them what high quality code review looks like. Second, there is a large number of inexperienced students along with a small number of experienced staff members. Code review in the classroom needs to account for and leverage this as best as possible. Finally, the goal of 6.031 is to teach students how to be great software engineers—this is much more of a learning experience than industry. These differences are what Caesar hopes to account for in being designed for the classroom.

Caesar takes a unique crowdsourcing approach to code review, using the students in the class as the crowd [1]. In particular, this crowdsourced code review in the classroom is similar to novice crowdworkers assigned to a task where they have limited

experience with the subject matter. Students taking 6.031 do have programming experience, but few have experience with good software engineering principles, and no prior MIT class incorporates code review into its curriculum.

The previous Caesar system, before the work done in this thesis, is an effective way for students in 6.031 to participate in code review for each others problem set code. However, since comments are simply left in a comment box with no restrictions on content, the feedback that students provide only come from the knowledge they have accumulated by attending class and doing readings on the class website. 6.031 holds one class at the beginning of the semester that focuses on what to look for during code review and how to do it well. While this class is informative, students start participating in code review shortly after, less than two weeks into the semester. We hope to help students learn how to code review better, as well as provide feedback about how they are doing during the process. To do this, we take ideas from crowdsourcing about how to improve the quality of crowd work, particularly for novice crowdworkers.

This thesis contributes three additions to Caesar. First, we add more scaffolding and structure to the interface for writing new comments. Second, we implement an initial training period, or practice task, for each code review session, allowing students to gain more knowledge and training before they review their peers' code. Finally, we add an interface for staff members to be able to grade and provide feedback on student code review comments more easily. To evaluate the effects of these additions, we compare data from code review comments between problem sets within the same semester and across different semesters.

The remainder of this thesis explains the interface and implementation of this work. Chapter 2 summarizes related work in peer feedback and crowdsourcing. Chapter 3 introduces the new interfaces. Chapter 4 explains the implementation of practice tasks. Chapters 5 and 6 present evaluation on work done for this thesis, and a discussion of that evaluation. Chapter 7 presents future work for the system, and Chapter 8 is a conclusion.

Chapter 2

Related Work

2.1 Caesar and 6.031

Caesar is an online code review tool created for use within 6.031: Elements of Software Construction, a software engineering class at MIT. Caesar allows for the distributed review of student code by their peers, staff moderation of the review process, and an overall, more social way for students to give feedback to each other.

Caesar was first created to process student code into small pieces, distribute these tasks to reviewers, and provide a web interface for students to code review their peers' work [7]. Further work refined and developed algorithms to optimize how student and staff time is spent [8]. In addition, new tools were added—a comment searching tool that allowed reviewers to reuse their previous comments, as well as a code search tool for instructors to search for specific patterns in student code [4].

Caesar uses a crowdsourcing approach towards code review. Crowdsourcing in typically the process of having tasks accomplished by enlisting the services of a large group of people. The work done in crowdsourcing is typically more repetitive and usually does not require deep expertise in a particular field. In the Caesar context, this large group of people are students enrolled in the class, and the task is for students to provide useful feedback to their peers on problem set code.

The standard review process done in Caesar for 6.031 is repeated five times throughout 6.031, for each of the five problem sets, named ps0 through ps4. Each

problem set spans two weeks, with an alpha deadline at the end of the first week and the beta deadline at the end of the two weeks. Students participate in code review right after the alpha deadline. Students have several files from the problem set to review, and each file has several students that will review it. After the code review period has passed, the comments are released to the author of the code. The current interface supports writing of a comment that is attached to one line of code, or a range of consecutive lines of code. For each comment, other students who are assigned to review the same file can upvote, downvote, or reply. The author of the file can also comment on the comments they've received. In the beta milestone, the staff looks to see that the authors of the code have responded to their code reviews, either by making a change to address that review in their code, or by leaving a comment in the code or in Caesar explaining why a change was not made.

2.2 Feedback and Crowdsourcing

Peer assessment is a collaborative learning technique in which students evaluate each other's work, and this is the use case for Caesar in 6.031. To effectively scale such peer assessment, crowdsourcing lessons can be applied. In particular, well-crafted training exercises can be very effective in enabling novice crowdworkers to perform high quality work [5]. In fact, novice crowd workers with an expert rubric can produce feedback nearly as helpful as feedback from experts [9]. This result was obtained from a study on crowdworker review of design projects, where the study found that rubrics caused improved writing style in terms of directness, motivation, and clarity. While 6.031 is not a design class, we think that a good rubric will also improve the writing style of code review comments in Caesar, resulting in improved comment quality.

Another study on design critiques through crowdsourcing used a system called Critiki, which also explores using rubrics and scaffolding [2]. In a study using Critiki, researchers found that review tasks that were scaffolded resulted in higher quantity of quality critiques, that were longer overall. We hope to add more scaffolding and rubrics to increase quality of code reviews in Caesar.

2.3 Crowdsourcing Quality

While crowdsourcing feedback is a very scalable solution, one problem that comes with it is maintaining crowdsourcing quality. In particular, this is a concern for novice crowdworkers—in our case, the relatively inexperienced students in 6.031.

Many techniques exist to improve crowd work quality, including verifying worker outputs with other workers, having experts review a subset of work, and using majority consensus [1]. One other way is the ground truth method, which compares answers from crowdworkers to a gold standard, which are usually known answers to tasks that were inserted into the work [1]. Both CrowdFlower and Mechanical Turk use this method. In particular, CrowdFlower combines the use of an initial training period before work, along with gold standard questions sprinkled throughout work after the training period [6]. Mechanical Turk researchers refer to gold standard questions as explicitly verifiable questions, and their experiments showed the importance of having these types of questions in three ways: (1) opening work with these questions to verify worker answers and require workers to process content, (2) signaling to workers their work will be scrutinized, and (3) reducing invalid responses [3].

Chapter 3

Code Review Interface and Process

The changes to the Caesar code review process can be broken down into three components—a more structured way to write comments, an interface for staff to grade student comments, and an implementation of practice tasks before normal code review tasks. This chapter describes these three components.

3.1 Structured Comments

In order to add more structure to code review comments in Caesar, we decided to appeal to the three foundation principles of 6.031: Safe From Bugs (SFB), Easy to Understand (ETU), and Ready For Change (RFC). The new comment user interface includes three dropdown menus, one for each principle. Students can choose to tag their comment with one of the three principles, along with a + to signify that the code author did a good job writing code that follows this principle, or a − for the opposite. Each comment written is required to include at least one `+/- sfb/etu/rfc` tag.

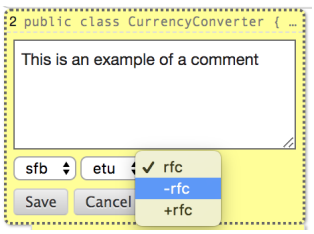

Figure 3-1: Caesar's new comment UI

After a student writes a comment, chooses at least one tag, and presses save, the system inserts the tag(s) used in a new line at the end of the comment, with multiple tags separated by one space. As a result, the tags used for each comment are saved directly into the text associated with that comment.

Figure 3-2: A written comment with three tags

While the user interface for comment replies does not require the usage of tags, we have seen some student incorporate this notation at the end of their comment regardless.

3.2 Grading Student Code Review

Another new component added to Caesar is the ability for lab assistants to grade the quality of student code review comments. The goal of this functionality is two-fold. First, we want to have a more systematic way of determining if a student put an adequate amount of effort into participating in code review for a particular problem set. Second, we want to use the data from LA grading of comments to see how the quality of student comments changes within the same semester and across the fall 2019 and spring 2020 semester.

The new lab assistant grading interface includes new +, ✓, and − buttons at the bottom of each comment. During each code review, lab assistants are assigned to a set number of files, and they grade each student written comment by clicking +, ✓, or −. + indicates the student did an excellent job providing a correct and high quality comment, ✓ means that the comment was sufficient, and − means the comment was low effort or gave incorrect advice.

Figure 3-3: A student comment from the LA view

The introduction of the LA grading interface also simplifies the process of grading student participation in code review. If a student has made an adequate number of check or plus graded comments, then we can automatically give them credit for participating in code review. As a result, less manual grading of participation needs to be done.

3.3 Practice Tasks

The most extensive addition to Caesar and 6.031 code review is the addition of practice tasks. Practice tasks occur at the beginning of a code review session, giving the student training exercises before they reviewed code from their peers. The code used in practice tasks is written by staff members and includes bugs or other problems related to 6.031 concepts. Each practice task has a set of expected answers, which are issues with the code that we expect students to identify with comments. Each expected answer consists of two major components—a range of line numbers at which to look for a student comment, and a list of keywords used to determine if the content of a student's comment identified the known issue.

3.3.1 Starting Code Review

After code review has opened, students start their code review tasks in their Caesar dashboard. On the first visit to the dashboard after the code review begins, students see a green "Start Reviewing" button.

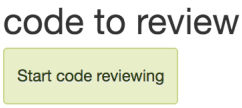

Figure 3-4: "Start Reviewing" button in Caesar

In the Fall 2019 semester and before, clicking the "Start Reviewing" button would display a list of tasks assigned to the student. Unopened task names are in bold.

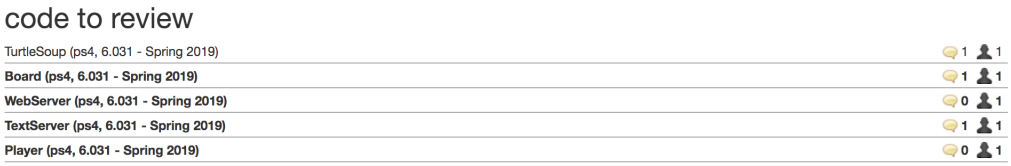

Figure 3-5: Old Caesar interface - list of tasks

However, following prior research discussed in section 2.3, one of the goals of practice tasks is to provide an initial training period for students. As a result, students would have to do the practice task first, prior to their regular code review tasks on other students' code. In order to ensure this happens, in the Spring 2020 semester, the "Start Reviewing" button opens the first task directly, which is the practice task. When going back to their dashboards, students no longer see a list of tasks—instead, they see a green "Continue Reviewing" button. Clicking that button re-opens the current unfinished code review task. The student can also see how many remaining code review tasks they have to do to the right of the button.

Figure 3-6: "Continue Reviewing" button in Caesar

3.3.2 Practice Task Workflow Overview

The workflow of a practice task consists of three steps.

First, when students open a practice task, they are greeted with special instructions that explain what a practice task is and what steps to follow. In this stage, the student cannot click the "Next" button yet. They proceed to look over the provided practice task code and make comments to point out problems with the code as they see fit.

Figure 3-7: Practice task: instructions and initial interface

After students make at least one comment on the practice task file, they can click "Next", which is now enabled.

Figure 3-8: Practice task: student makes a comment

Clicking "Next" results in automated replies and new comments on the file. These automated comments are generated based on where the student left comments and if those comments contained certain keywords. Each of the expected answers will result in at least one feedback comment. Possible outcomes for these feedback comments are:

1. The student did not make a comment in the expected line range, so the system gives feedback in a new comment that explains what the problem was and what the expected keywords were.

2. The student made a comment in the expected line range, but they did not use any of the expected keywords. The system will reply to such comments and explain what the problem was and what the expected keywords were.

3. The student made a comment in the expected line range and used at least one

of the expected keywords. The feedback comment explains what the problem with the code was, and it also confirms which expected keywords the student used.

In the figure below, we can see an example of a practice task comment in the expected line range, using two expected keywords (outcome 3 above).

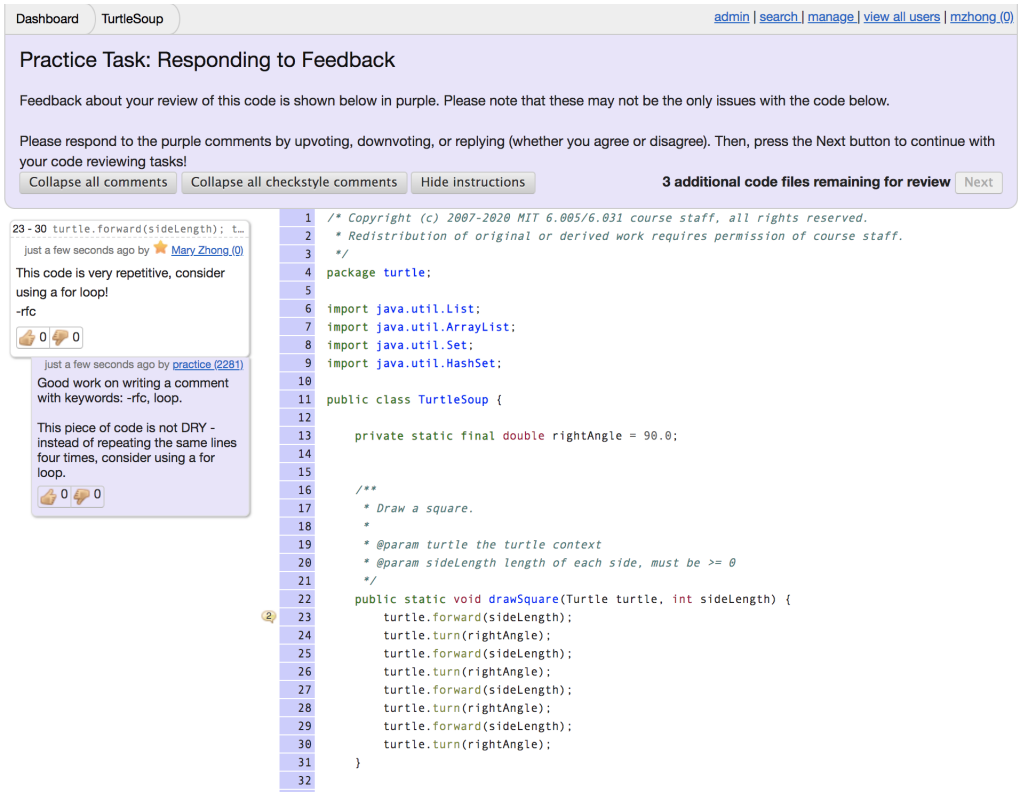

Figure 3-9: Practice task: automated feedback

Finally, students look over the feedback comments and upvote, downvote, or reply to at least one. In this way, we can collect information about whether students agreed or disagreed with the feedback comment.

Chapter 4

Practice Task Implementation

The implementation of practice tasks in Caesar uses the Django framework, with a Python backend and javascript with JQuery, HTML, CSS in the frontend. This section explains the different components of the practice task implementation in more detail.

4.1 Backend Models

A Django model is the source of information about some aspect of stored data, containing fields and other information about that data. Before we implemented practice tasks, Caesar already had several Django models used for code review.

The following are existing models within Caesar:

- `Subject`: A subject or class. For this thesis, the `Subject` is 6.031.
- `Semester`: A semester for a subject. For this thesis, the relevant semesters are 'Fall 2019' and 'Spring 2020'.
- `Assignment`: Each semester contains several assignments. In this case, we have ps0 through ps4, for a total of five assignments per semester.
- `Milestone`: Each assignment has three milestones of two types: `SubmitMilestone` and `ReviewMilestone`. In 6.031, each problem set assignment is split into

an alpha and beta submission, for a total of two submit milestones. The code review process is tied to one review milestone per assignment.

- `Submission`: Each milestone has many submissions attached to it, one per student.
- `File`: Each submission contains files. The `File` model contains the content of the file, as well as data about the file, such as file path and which submission it is from.
- `Chunk`: Each file can be split into multiple chunks. However, currently in 6.031, there is one `Chunk` for each `File`.
- `Task`: Finally, a `Task` is a single task that a student receives on their dashboard for a code review period. Each `Task` contains fields for the `Submission`, `Chunk`, and `Milestone` for its contents. Each task also has one `Reviewer`. There can be many tasks associated with the same `Chunk`, and each reviewer has many tasks.

As mentioned above, the code review process is tied to instances of the `ReviewMilestone` model. As a result, the practice task implementation builds off of review milestones. To integrate practice tasks into the backend, we have created three new models: `PracticeProblem`, `ExpectedAnswer`, and `PracticeTask`. These three models were implemented and deployed for the Spring 2020 semester.

4.1.1 `PracticeProblem`

A practice problem contains the information needed for one practice task file. As a result, it has the following fields:

- `name`: The name of the practice problem, as a String. This is used internally only for identification purposes and not shown to students.
- `file`: The file used for this practice problem. This field is a foreign key to an instance of the `File` model.

- `milestone`: The review milestone for this practice problem. For example, if this was a practice task intended for ps3, the `milestone` field would point to the `ReviewMilestone` for ps3.

Every `ReviewMilestone` for the five assignments in the Spring 2020 semester contained at least one `PracticeProblem`.

4.1.2 ExpectedAnswer

An expected answer represents an issue in a practice task file that the student is expected to leave a comment about. There is at least one `ExpectedAnswer` associated with each `PracticeProblem`. Here are the fields:

- `keywords`: Comma separated keywords that students are expected to use in their comment.
- `comment`: A comment that identifies and explains the problem the student is expected to comment on. This comment goes into every feedback comment generated for this expected answer. More detail on the content of the generated feedback comments will be discussion in section 4.3.
- `lines`: A comma separated, no whitespace, list of line numbers and/or line number ranges where a student comment is expected. A line number range is in the format `a-b`, where `a` is the starting line number and `b` is the ending line number of the range. An example of a value that this `lines` field could be is `"4,10-12,8"`, meaning that a comment is expected that starts on lines 4, 8, 10, 11, or 12.
- `practice_problem`: The `PracticeProblem` instance that this expected answer is associated with.

4.1.3 PracticeTask

Finally, the third model introduced for this thesis is `PracticeTask`. `PracticeTask` is actually a model that inherits from the `Task` model. As a result, it contains all

the fields that a task does, but has some additional ones:

- `practice_status`: The current status of the practice task. This field can have a value of one of the four statuses:
    1. `New`: No comments have been written by the student.
    2. `Commented`: The student has left at least one comment on the practice task, but they have not pressed "Next" to receive feedback.
    3. `Feedback`: The student has received automated feedback comments for their practice task comments.
    4. `Done`: The student has responded to the feedback by upvoting, downvoting, or replying to at least one automated feedback comment.
- `practice_problem`: The practice problem for this practice task.

4.2 Practice Task Set Up

In order to set up a practice task, a script is used to create practice problems. The scripts takes in Java files that contain the content for practice tasks, in addition to some metadata. This metadata includes names used for the created practice problems, the paths to the files, and the id of the review milestone these practice problems are for. Each input file results in one created practice problem instance.

The input files to this script are normal Java files along with some metadata within. First, there is a JSON style comment at the top of the file that lists the number of expected answers for this file and their corresponding name identifiers, keywords, and feedback comment. Second, the file has to identify where the expected comments students make should be located. As explained in section 4.1.2, each expected answer is associated with some lines or line ranges. The input files to this setup script identify those lines by including the appropriate name identifier in a comment at the beginning of corresponding lines.

Below is an example of a portion of the one input file used for ps0 practice tasks in the Spring 2020 semester:

Listing 4.1: ps0 practice task input file for Spring 2020

```
/*[
{
    "name": "DRY",
    "keywords": ["-sfb", "-rfc", "DRY", "repeat", "loop"],
    "comment": "This piece of code is not DRY - instead of repeating the same lines four
        times, consider using a for loop."
},
{
    "name": "magicNumber",
    "keywords": ["-etu", "-sfb", "magic"],
    "comment": "This method uses the magic number, 360, twice. One solution could be to
        treat it similarly to `rightAngle`, or make it a final, local variable."
},
{
    "name": "final",
    "keywords": ["-sfb", "final"],
    "comment": "The variable `sideLength` could be final since it is never reassigned."
},
{
    "name": "varName",
    "keywords": ["-etu", "variable name"],
    "comment": "The variable `a` should have a more descriptive name, such as `
        angleRadiansHalved`."
}
]*/
/* Copyright (c) 2007-2020 MIT 6.005/6.031 course staff, all rights reserved.
 * Redistribution of original or derived work requires permission of course staff.
 */
package turtle;

import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;

public class TurtleSoup {

    private static final double rightAngle = 90.0;


    /**
     * Draw a square.
```

```
40       *
41       * @param turtle the turtle context
42       * @param sideLength length of each side, must be >= 0
43       */
44      /*DRY*/public static void drawSquare(Turtle turtle, int sideLength) {
45          /*DRY*/turtle.forward(sideLength);
46          /*DRY*/turtle.turn(rightAngle);
47          /*DRY*/turtle.forward(sideLength);
48          /*DRY*/turtle.turn(rightAngle);
49          /*DRY*/turtle.forward(sideLength);
50          /*DRY*/turtle.turn(rightAngle);
51          /*DRY*/turtle.forward(sideLength);
52          /*DRY*/turtle.turn(rightAngle);
53      }
54
55      /**
56       * Determine the length of a chord of a circle.
57       * (There is a simple formula; derive it or look it up.)
58       *
59       * @param radius radius of a circle, must be > 0
60       * @param angle in degrees, where 0 <= angle < 180
61       * @return the length of the chord subtended by the given `angle`
62       *         in a circle of the given `radius`
63       */
64      public static double chordLength(double radius, double angle) {
65          /*varName*/final double a = Math.toRadians(angle) / 2.0;
66          /*varName*/return 2 * radius * Math.sin(a);
67      }
68
69      /**
70       * Approximate a circle by drawing a many-sided regular polygon,
71       * using only right-hand turns, and restoring the turtle's
72       * original heading and position after the drawing is complete.
73       *
74       * @param turtle the turtle context
75       * @param radius radius of the circle circumscribed around the polygon, must be > 0
76       * @param numSides number of sides of the polygon to draw, must be >= 10
77       */
78      public static void drawApproximateCircle(Turtle turtle, double radius, int numSides) {
79          /*magicNumber*/final double exteriorAngle = 360 / numSides;
80          /*final*/int sideLength = (int) Math.round(chordLength(radius, exteriorAngle));
81          for (int i = 0; i < numSides; i++) {
82              turtle.forward(sideLength);
```

```
83              turtle.turn(exteriorAngle);
84          }
85      }
```

In the file above, lines 1 through 22 (colored red) contain the JSON metadata about this practice task. There is a list of four objects, and each object contains the information need for one expected answer. In this file, there are four expected answers, and there are named `DRY`, `magicNumber`, `final`, and `varName`. The first expected answer has five keywords, and has a comment that explains that the code repeats the same lines, which is not DRY (Don't Repeat Yourself).

Next, the lines that are relevant for each expected answer begin with an in-line comment containing the name. For example, lines 44 to 52 start with `/*DRY*/`, and those are the lines where students are expected to make a comment about repetitive code. The same formatting can be seen for the other three expected answers.

After the script processes input files, it creates one practice problem instance for each file. Each practice problem instance may have multiple expected answers. In the case of running the script with the file in listing 4.1 above, the system will create one practice problem with four expected answers.

Finally, to finish the process, the JSON metadata and in-line name comments are deleted, and the remaining parts of the file get put into a new `File` instance. This is the file that the `file` field of the created `PracticeProblem` will point to (section 4.1.1).

4.2.1 Practice Task Assignment

After the implementation of the necessary backend models needed for practice tasks and writing a script for creating practice problems and expected answers, the existing Caesar task assignment process needed to be modified to assign practice tasks.

Each `ReviewMilestone` has a `student_count` field, which is an integer representing how many code review tasks a student would receive for that code review milestone. To assign practice tasks, a new field `practice_count` was added—the number of practice tasks each student would receive. The total number of tasks is

still `student_count`, so this means that (`student_count` − `practice_count`) is the number of regular tasks received that contain code from students' peers. The creation, assignment, and distribution of those regular tasks are existing parts of Caesar.

Now that Caesar knows the correct number of regular and practice tasks to assign, it has to choose which practice tasks. For review milestones that only have one practice problem, all practice tasks created and assigned use content from that same practice task file. If there are multiple files and practice problems, then Caesar randomly chooses `practice_count` number of them to assign, or chooses all of them if there are fewer than `practice_count` practice problems.

4.3 Practice Task Flow and Automated Feedback

As mentioned in section 4.1.3, each practice task has multiple states associated with it—`New`, `Commented`, `Feedback`, `Done`. These states change as a student goes through the intended practice task flow. Section 3.3.2 gave an overview of this workflow, and this section will provide more information into the implementation and details of this flow.

1. When the practice task is first opened, it is in the `New` state. The student has made no comments yet. Even if they exit the task and click "Continue Reviewing", the task will continue to be in this state. In this state, the "Next" button is grayed out and disabled. The purpose here is to ensure students make at least one comment on the practice task.

2. Once the student makes a comment, the practice task will transition to the `Commented` state. In this state, the "Next" button is no longer disabled, and students can choose to make more comments or to click the button and proceed to the next stage of the practice task flow.

3. After the "Next" button is clicked, the practice task enters the `Feedback` state. The instructions at the top of the task change, and the background color of that

area changes to purple. Most importantly, this is the state where automated feedback comments are generated. They appear as replies or new comments on the practice task, and they also have a purple background color.

To generate these feedback comments, when the student clicks "Next" from the `Commented` state, Caesar searches through all of their comments on that practice task file. Then, for each expected answer for that practice task, the system looks for student comments in that expected answer's line range(s).

For comments in that range, the system looks for the expected answer's keywords in the comment text. If the comment includes at least one keyword, then the feedback comment generated is a reply to the student comment and gives positive feedback on the choice of keywords. If no expected keywords are found, the system generates a feedback comment as a reply, with text that tells the student what the appropriate keywords are.

If there is not a student comment in the expected line ranges for an expected answer, the system generates a feedback comment that is inserted as a new comment to the file at the expected location. This comment informs the student that there was an expected comment here and explains what the issue with the code is.

All three types of feedback comments will include text about what the problem with the code was. This text is the same as the `"comment"` value in the JSON metadata in the practice task file in section 4.2. It is also the same text stored in the `comment` field of `ExpectedAnswer` objects (section 4.1.2). The three types of feedback comments are shown in the figures 4-1, 4-2, and 4-3 below.

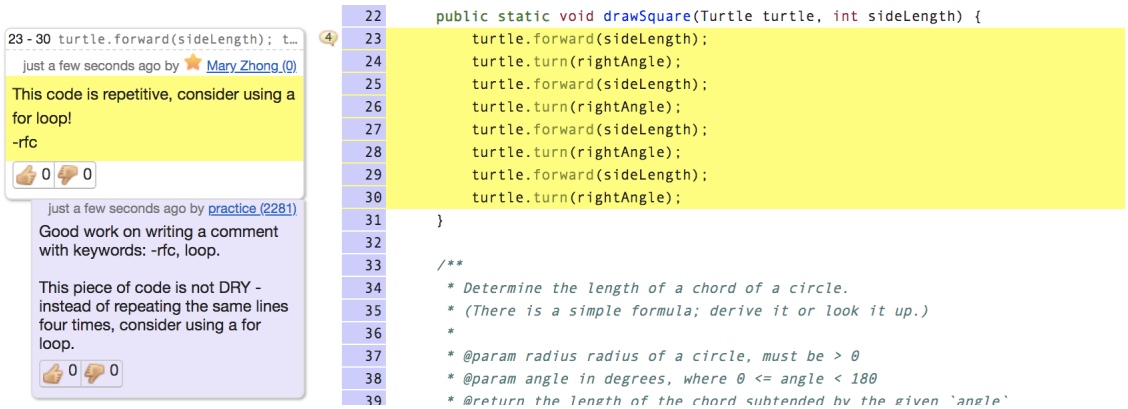

Figure 4-1: Automated feedback: at least one expected keyword

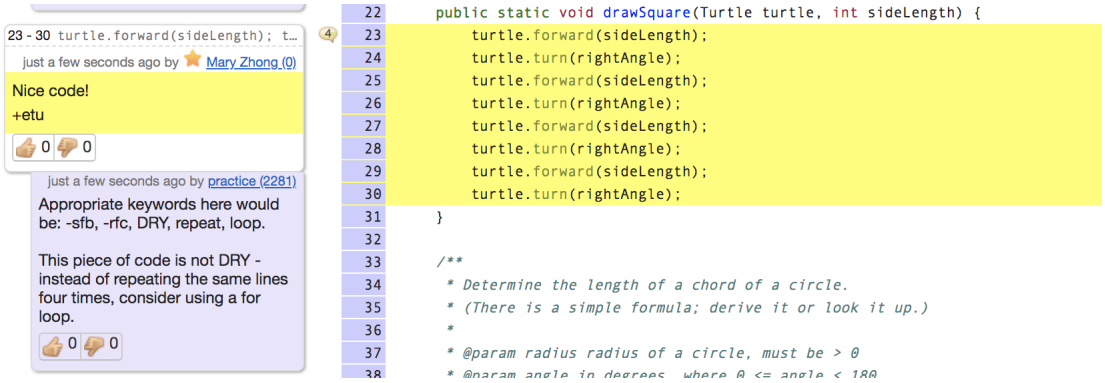

Figure 4-2: Automated feedback: no expected keywords

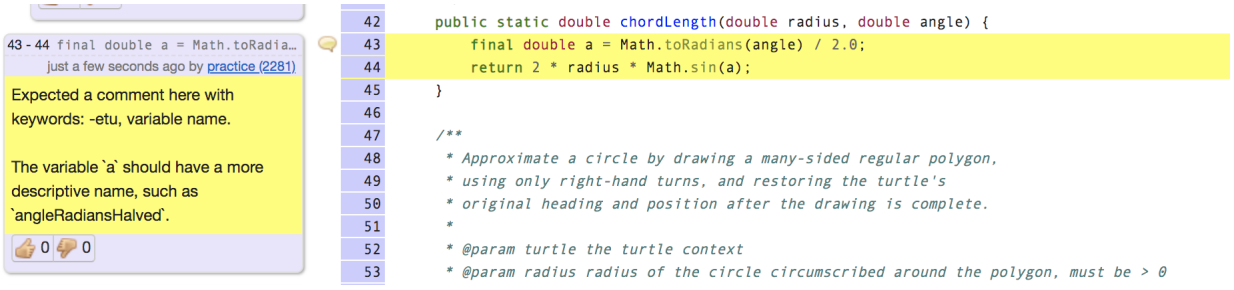

Figure 4-3: Automated feedback: missing comment

4. Finally, after the student reads over the feedback comments, they can follow the instructions to upvote, downvote, or reply to at least one feedback comment. This gives the student an opportunity to disagree or agree with the feedback they were given.

4.4 Fall 2019 Semester

Practice tasks were partially introduced for ps2 in the Fall 2019 semester. For this problem set, students were assigned one practice task. However, implementation of automated feedback comments was not complete. As a result, comments about the problems with the code in the practice task were created and added to all of the practice task files after the code review period. However, we do not know how many students reopened their code review task and saw these comments. For ps3 and ps4 in the fall, automated feedback was implemented. After making comments on a practice task, students were able to click "Next" and receive purple colored feedback comments. The sections below explore the main differences between the fall implementation of practice tasks and the current Spring 2020 one.

4.4.1 Assigning Practice Tasks

In the Fall 2019 semester, Caesar assigned practice tasks by always assigning one out of the possible practice task files. In the fall, the `practice_count` field of review milestones did not exist yet, and we defaulted to one practice task per problem set code review. This one task was also picked randomly from all the available practice task files.

4.4.2 Generating Automated Comments

As explained in section 4.3, the practice task workflow includes a `Feedback` state, where students can see automatically generated feedback on comments they have made on practice tasks. These feedback comments are generated based on whether the student comment contains any of the expected keywords.

However, in the Fall 2019 semester, the backend models described in section 4.1 did not exist yet, and the keywords used were only `+/- sfb/etu/rfc` tags. Since we did not have the models to store information about practice problems and expected answers, this information was recorded as comments on the original practice task files.

The practice task files in Fall 2019 were uploaded to Caesar as a submission by a

user with username `practice`. Then, we logged in as the `practice` user and wrote comments on each of the files. Each comment represented one expected answer. The comment text, excluding the tags used, is the text explaining what the problem with the code is. This is the same as the `comment` field of `ExpectedAnswer` objects that were used in Spring 2020. The tags used in these comments were the expected tags. The range of lines that the comment was left on are the expected starting lines for student comments.

When a student made a comment on the practice task, the automated feedback for that student comment was generated based on the comments by the `practice` user on that original file. Below, we can see an example of one `practice` user comment on a practice task file for ps3 in Fall 2019.

Figure 4-4: Practice task file from Fall 2019

The one comment in figure 4-4 means that there is an expected student comment starting between lines 17 and 21. This comment has expected keywords/tags of `-sfb` and `-etu`. The correct content of this comment should have pointed out missing types in the datatype definition. Therefore, if a student that received this practice

task made a comment starting somewhere on lines 17-21 and used at least one of the two expected tags, they would have received a positive feedback comment. If they did not use any of the expected tags, the feedback comment would inform them of the appropriate tags. Finally, if they did not leave a comment in that range at all, the system would inform the student that there was an expected comment in that range. These three outcomes are the same as Spring 2020.

Chapter 5

Evaluation

This section discusses the process of deploying the new comment interface, LA grading interface, and practice tasks in 6.031 over the Fall 2019 and Spring 2020 semesters. Then, we explore the two areas in which we evaluate the result of our work.

5.1 Spring 2020 Semester

When looking at the data from the previous two semesters, we first have to consider the special circumstances for the Spring 2020 semester.

In December 2019, SARS-CoV-2, the virus causing the disease Covid-19, emerged in the city of Wuhan, China. By February 2020, it was clear that this virus was spreading in the United States. By March 2020, the situation escalated, and on March 10, 2020, MIT announced that undergraduates would be required to move out of campus dorms and MIT-affiliated housing by March 17, and that classes would be completely online starting March 30, after spring break. Out of the 4530 undergraduates enrolled at MIT, 3345 students live on campus dorms and about 1000 students live in MIT-affiliated housing. As a result, 95% of undergraduates were affected. A little above 400 students remained on campus as the result of petitions, and even they likely had to move to consolidated dorms.

This news was clearly very disruptive to the community. Since 6.031 is an undergraduate course, almost all of our students were affected. Code review for ps2, ps3,

and ps4 were done after spring break, when the course was fully online. Covid-19 and having campus life cut short affected our students in a significant way, and this should be taken into account when looking at data from the Spring 2020 semester.

5.2 Deployment in 6.031

The deployment of the changes in this thesis started with introducing `+/- sfb/etu/rfc` tags and the new LA grading interface at the beginning of the Fall 2019 semester. Also in the Fall 2019 semester, as explained in section 4.4, practice tasks were implemented partially in ps2, and then automated feedback was implemented for ps3 and ps4. For the Spring 2020 semester, practice tasks were implemented with the current system for all problem sets.

After examining practice task data from ps3 and ps4 in the fall, we realized a few major flaws in the system. First, the feedback system needs to be able to check for keywords beyond `+/- sfb/etu/rfc` tags. While those tags are a good way to give a high level idea of what the comment is about, they do not distinguish beyond the three principles of 6.031. Second, the feedback system needs to be able to check for expected comments in disjoint line ranges. For example, we found many cases of students commenting on an issue in the practice task code in a location we did not anticipate, but that still made sense. We want to check for such a comment in two locations of the file that are not adjacent. Finally, there needs to be state management for a practice task file. Students should be able to leave a practice task any time and return to it as if they had never left. In the Fall 2019 semester, if students made a comment on a practice task file and left without clicking "Next" to receive feedback, they never received feedback comments, since Caesar would mark that task as done. There was no way to distinguish between practice tasks and regular tasks in the backend.

The changes explained in section 4.1 solve the above problems. Section 5.4 goes into more detail about the types of analysis we performed to come to the conclusions from above.

5.3 Code Review Participation

As explained in 2.2, one of the goals of adding tags to comments is to create a more scaffolded comment structure, hopefully increasing the quality and quantity of code review comments in Caesar. In addition, we hoped that practice tasks would serve as an initial training period, resulting in giving students more code reviewing experience. More experience would hopefully result in higher quality code review comments by students in 6.031.

In order to evaluate the effect of structured comments and practice tasks on comment quality and quantity, we look at data on student comment length and quantity, LA grading of student comments, and comment data from the previous semester before Fall 2019.

5.3.1 Quantity of Comments

We first look at the quantity of comments made in the Spring 2019, Fall 2019, and Spring 2020 semesters. Below are three tables displaying the number of students, number of regular tasks each student got, total number of comments, total number of characters, average characters per comment, and average comments per student per task for all five problem sets, over the past three semesters. These numbers do not include comments made on practice task files, since we want to evaluate the effect of practice tasks on regular code review tasks.

	tasks	students	comments	chars	chars/comment	comments/student/task
ps0	3	241	3587	332869	92.80	4.96
ps1	6	228	4103	403242	98.28	3.00
ps2	6	222	3952	344737	87.23	2.97
ps3	6	218	3337	294383	88.22	2.55
ps4	6	204	2858	248011	86.78	2.33

Table 5.1: Comment quantity: Spring 2019 semester

	tasks	students	comments	chars	chars/comment	comments/student/task
ps0	3	163	2270	204525	90.10	4.87
ps1	10	163	3335	306591	91.93	2.17
ps2	5	153	2494	226066	90.64	2.99
ps3	5	144	2276	204500	89.85	2.89
ps4	5	140	2252	181071	80.40	2.78

Table 5.2: Comment quantity: Fall 2019 semester

	tasks	students	comments	chars	chars/comment	comments/student/task
ps0	2	223	3377	273495	80.99	6.14
ps1	9	219	5020	424578	84.58	2.55
ps2	5	209	3487	288285	82.67	3.33
ps3	5	190	2703	226609	83.84	2.85
ps4	5	185	2810	222939	79.34	2.97

Table 5.3: Comment quantity: Spring 2020 semester

Since `+/- sfb/etu/rfc` tags were introduced in Fall 2019, for the character counts in Fall 2019 and Spring 2020, we removed the number of characters that were part of those tags in order to make the numbers comparable to Spring 2019. Below are graphs of chars/comment and comments/student/task for the three semesters.

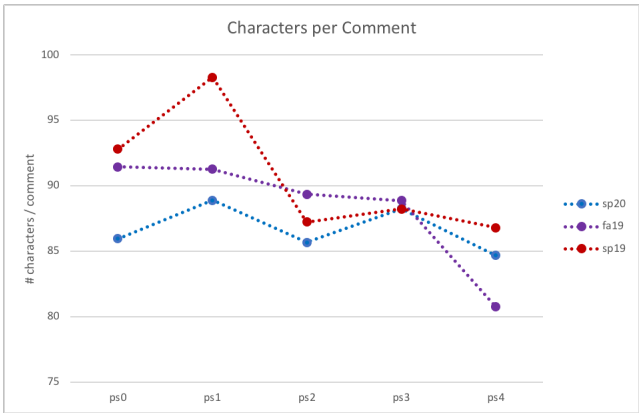

Figure 5-1: Characters per comment - sp19, fa19, sp20

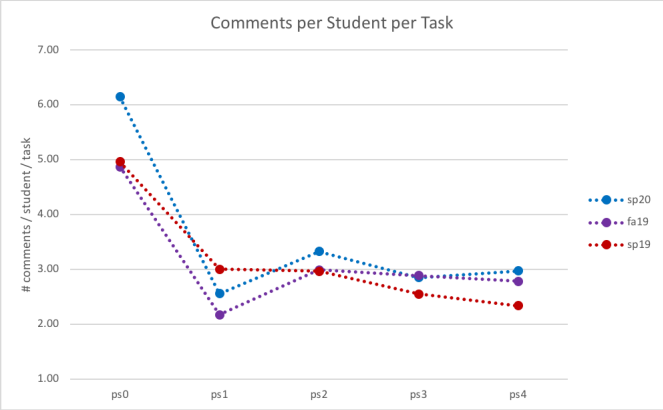

Figure 5-2: Comments per student per task - sp19, fa19, sp20

To evaluate the effect of the introduction of tags and practice tasks, we can look at the number of comments students write per code review task. We can also look at the length of those comments. From our experience looking at past student comments, longer comments seem to contain better and more useful content. In addition, shorter comments tend to be vague critiques such as "this code is confusing" or short compliments such as "nice code!".

Figure 5-1 shows the average characters per comment for the five problem sets for the past three semesters. We see here that the Fall 2019 semester seems to have higher characters per comment compared to the Spring 2020 semester, except for ps4. In addition, the Spring 2019 data has the highest average characters per comment for all problem sets except for ps2 and ps3—but the values for those two problem sets are all within 5 characters, and it is not clear how significant that difference is. One anomaly to note is that the number of regular (non-practice) tasks students were assigned for problem set 1 in Spring 2019 was only 6 tasks, whereas that number jumped to 10 and 9 tasks, for Fall 2019 and Spring 2020 respectively. One hypothesis could be that students saw that they had a lot of tasks in Fall 2019 and Spring 2020, and therefore chose to write shorter comments on average, wanting to finish code reviewing faster. From this graph, however, it does not seem like the usage of practice tasks in ps0 and ps1 for Spring 2020 increased character counts for student comments.

Figure 5-2 shows the average number of comments per student per task for all

five problem sets over the past three semesters. First, we notice that the value for ps0 in Spring 2020 is higher than the other two semesters—the average number of comments per task each student made was over 1 higher than Fall 2019 and Spring 2019. This seems to suggest that having a practice task for ps0 in Spring 2020 resulted in that higher value. The practice task for ps0 in Spring 2020 consisted of one file with four expected answers, so every student received the same practice task. The expected answers for ps0 focused on basic coding practice principles, which can be found in one of the 6.031 readings: `http://web.mit.edu/6.031/www/sp20/classes/04-code-review/`. The intention is for students to see examples of common issues regarding general principles of good coding. Then, they would be able to comment on similar issues on their peers' code. The data here supports that ps0 in Spring 2020 was successful in doing this. Students are also mostly new to Java and good coding principles when they work on ps0, and that might result in more issues with student code that other students comment on during code review. In addition, the focus of the ps0 practice task is very defined—the issues presented in expected answers are reinforced heavily in class. As a result, we think that students have a clearer idea on what issues to look for in their peer's code, after being in class and doing the practice task.

For the rest of the problem sets, the values from Spring 2020 tend to be higher than those of Fall 2019, except for a little lower for ps3. However, these differences are very small. Looking at the Spring 2019 data, we do see that the data points for ps3 and ps4 are lower than the other two semesters. This supports that practice tasks for ps3 and ps4 in the later two semesters increased the number of comments; however, these numbers may be too similar to come to that conclusion.

Figure 5-2 also shows that Spring 2019 has the highest number of comments per student per task for ps1. As mentioned before, students had the fewest ps1 code review tasks in Spring 2019 compared to the other two semesters. This supports our earlier hypothesis—students saw that they had a lot of tasks Fall 2019 and Spring 2020, and therefore chose to write fewer comments on average, wanting to finish code reviewing faster.

Finally, going back to the concerns raised in section 5.1, it does not seem like Spring 2020 is an outlier in this set of data. We do see in table 5.3 that the amount of participation for ps2, ps3, and ps4 dropped, with only 185 students out of the 231 enrolled participating in code review for ps4. Tables 5.2 and 5.1 show that the number of students participating dropped as the semester went on in Spring 2019 and Fall 2019, but the drop was steeper for Spring 2020. It is hard to tell exactly how Covid-19 affected the semester, but other than the drop in student participation, we did not see a significant decrease in comment quantity compared to past semesters.

5.3.2 Quality of Comments

In addition to seeing how the quantity of comments and characters changes over the past three semesters, we also investigate how the quality of comments changes. As explained in section 3.2, the Fall 2019 semester introduced a new interface for LAs to grade the quality of student code review comments. This functionality was not available in the Spring 2019 semester, so this section will only compare the past two semesters.

Due to a limited number of lab assistants on the 6.031 staff, not all student comments were able to be graded. Below are tables that show what percentage of student comments that were graded received $+$, ✓, and $-$ grades. Similar to the previous section, this data is based on student comments on normal code review tasks, not practice tasks.

	$+$	✓	$-$
ps0	23%	73%	4%
ps1	24%	73%	3%
ps2	26%	67%	8%
ps3	21%	73%	6%
ps4	19%	78%	3%

Table 5.4: LA comment grading: Fall 2019 semester

	+	✓	−
ps0	26%	69%	5%
ps1	22%	73%	5%
ps2	24%	71%	5%
ps3	26%	69%	4%
ps4	22%	74%	4%

Table 5.5: LA comment grading: Spring 2020 semester

In order to compare between semesters, we create three graphs to compare the percentages of each type of grade.

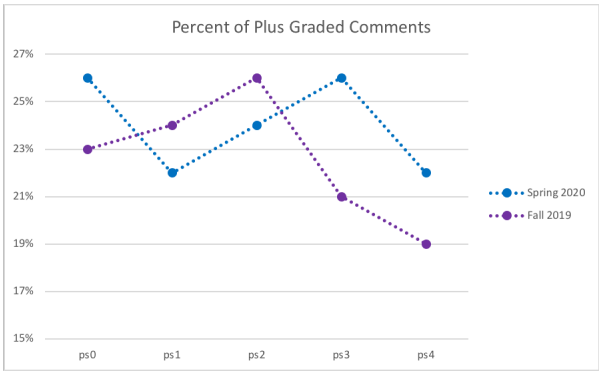

Figure 5-3: Plus graded comments

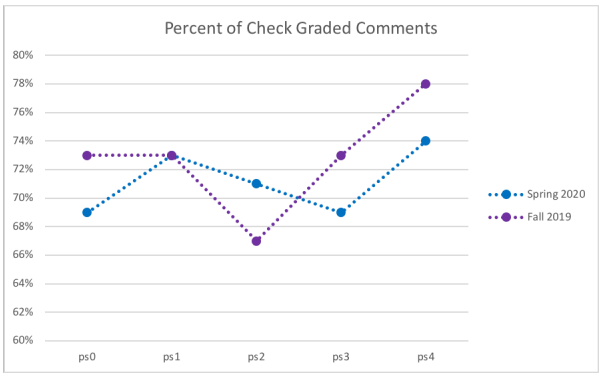

Figure 5-4: Check graded comments

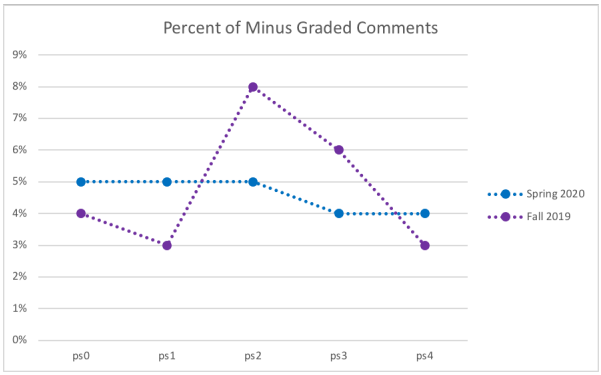

Figure 5-5: Minus graded comments

Since ps0, ps1, and ps2 of Spring 2020 included practice tasks with automated feedback, and Fall 2019's did not, we first take a look at those problem sets. While ps0 for Spring 2020 had more plus graded comments than Fall 2019, the opposite was true for ps1. For check graded comments, the Spring 2020 had a lower percentage. For minus graded comments, the Spring 2020 semester had a slightly higher percentage. In fact, looking at all the data points in figure 5-5, we note that they're within 2 percentage points of each other, except for Fall 2019's ps2 value, at 8%. It is unclear why this value is an outlier compared to the others. As explained in section 5.3.1, ps0 code review had clearly defined issues that students could focus on. This could explain a higher percentage of plus graded comments for ps0 in Spring 2020—students identified potential bugs and coding principle issues well. This also adds to evidence that the ps0 practice task was effective in teaching students what to look for during ps0 code review.

Looking at ps3 and ps4, we see that Spring 2020 had a higher percentage of plus comments compared to the fall. Similarly, the fall has fewer check comments compared to the spring. Both of these problem sets had the automated practice task feedback, but this suggests that the changes to the practice task system from fall to the spring helped students write higher quality comments for regular code review tasks.

Finally, regarding section 5.1 and comment quality, it also does not seem like the Spring 2020 was an outlier. In figure 5-3, the Spring 2020 semesters had more

problems sets with a higher percentage of plus graded comments compared to Fall 2019. For check graded comments, figure 5-4 also does not show that either semester had clearly lower percentages than the other. Last, the Spring 2020 comments did not see an increase in the percentage of minus graded comments as the semester progressed, as shown in figure 5-5.

5.4 Practice Task Accuracy

In addition to seeing how practice tasks and tags might have effected the quantity and quality of student code review comments, we want to analyze the accuracy of the feedback given in practice tasks. In order to do this, we divide feedback comments given in the following cases:

- **true positive**: The student *did comment* in the correct line range with at least one expected keyword, the feedback system gave *positive* feedback, and the content of the comment was *correct*.

- **false positive**: The student *did comment* in the correct line range at least one expected keyword, the feedback system gave *positive* feedback, but the content of the comment was *incorrect*.

- **true negative**: The student *did not comment* in the correct line range, may have used some expected keywords, the feedback system gave *negative* feedback, and the content of the comment was *incorrect*.

- **false negative**: The student commented in the correct line range with no expected keywords, *or* they commented outside of the correct line range, the feedback system gave *negative* feedback, but the content of the comment was *correct*.

We hope that the feedback system maximizes true positives and negatives, while minimizing false positives and negatives. In particular, we believe false negatives

are the worst errors to occur—the student actually made a correct comment and identified the problem with the code, but they received negative feedback.

To perform analysis on the accuracy of feedback given in practice tasks, we looked at all student comments on all practice tasks with automated feedback in the Fall 2019 and Spring 2020 semester. This means ps3 and ps4 for the fall and all problem sets for the spring. To be able to detect true and false positives and negatives, we went through all practice task comments and marked (1) which expected answer topic they were related to (if any) and (2) whether the comment content was correct. After annotating the data with topics and comment correctness, we counted the number of true positives and false positives and negatives.

5.4.1 Fall 2019

In the Fall 2019 semester, we only had automated feedback for ps3 and ps4. In addition, the expected keywords for practice task comments were only `+/-` `sfb/etu/rfc` tags.

	problem set:	ps3	ps3	ps3	ps3	ps3	ps3	ps3	ps3
	file:	ex1/Expression.java	ex1/Expression.java	ex2/Expression.java	Filename.java	Horizontal.java	Horizontal.java	Horizontal.java	Resize.java
	line range:	17-21	68-73	59-63	14-15	9-10	19-20	35-56	21-22
	expected answer topic:	datatype_def	flip	size	filename_AF	fields	sre	flip	ri
	expected tags:	-sfb -etu	-sfb -etu	-sfb -etu	-sfb -etu	-sfb	-sfb -etu	-sfb	-sfb
	# students with right content	17	7	16	14	7	2	16	14
	# students with wrong content	6	2	1	7	0	0	2	4
true positive	**right content exact tags within range**	2	4	7	6	2	1	13	11
		11.76%	57.14%	43.75%	42.86%	28.57%	50.00%	81.25%	78.57%
true positive	**right content missed tags within range**	14	3	9	8	1	1	0	2
		82.35%	42.86%	56.25%	57.14%	14.29%	50.00%	0.00%	14.29%
true positive	**right content added tag(s) within range**	1	0	0	0	4	0	3	1
		5.88%	0.00%	0.00%	0.00%	57.14%	0.00%	18.75%	7.14%
false negative	**right content no right tag(s) within range**	0	0	0	0	1	0	0	2
		0.00%	0.00%	0.00%	0.00%	14.29%	0.00%	0.00%	14.29%
false negative	**right content wrong line**	0	0	0	0	0	0	0	0
		0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
false positive	**wrong content one of right tags within range**	6	1	0	4	0	0	1	2
		100.00%	50.00%	0.00%	57.14%	n/a	n/a	50.00%	50.00%

Figure 5-6: ps3 practice task accuracy - fa19

Figure 5-6 above is split into 10 columns. The eight columns on the right that have ps3 in the top cell each represent the data for one expected answer. As we can see, ps3 had eight expected answers, split over 5 files (the second row of the table).

The figure is also split into 5 horizontal sections, separated by a grayed out row.

1. The first section lists out the metadata about the practice task and expected answers. This includes the problem set number, the practice task file name, the line range for the expected answer, the topic/name of the expected answer, and the expected tags.

2. The second section lists out how many students had comments with the right content and wrong content, regardless of location of the comment or the tags it used.

3. The third section shows the number of students with a comment that was a true positive. This is broken down into three cases—students that used the exact

expected tags, students that used extra tags, and students that *missed* at least one expected tag (but still used at least one expected tag). Since comments had to be in the expected range to be given feedback, all of the numbers in this section come from students who made comments in the expected line range for that expected answer.

4. The fourth section lists out data on the false negatives. There are two cases here: (1) students with a comment that did not use any of the expected tags but was in the right line range, and (2) students with a comment that used at least one expected tag but was not in the expected line range. Since these are false negatives, comments in either case *did* have the right content.

5. Finally, the fifth and final section gives data on false positives. In this case, these numbers are students with a comment that had one of the expected tags and was in the right line range, but the comment *did not* have the right content.

The percentages in the table take the number of students in each category and divides it by the appropriate number in horizontal section number two. For example, for the first expected answer in ps3 with topic `datatype_def`, exactly 2 students made comments with the right content, with the exact expected tags, and in the right range. This means that $\frac{2}{17} = 11.76\%$ of students who got the right content fall into this category.

The last horizontal section (false positives) is the only place where the denominator of the percentages is the # students with the wrong content. For the expected answer in the last column with topic `ri`, there were 2 students with comments that had the wrong content, but who received positive feedback because they used at least one of the right tags and were within the right range. Since there were a total of 4 students with wrong content for that expected answer, $\frac{2}{4} = 50\%$ of students who had wrong content were marked as false positives by the feedback system.

Since the goal is to minimize false positives and false negatives, we can take a look at the data in those two sections. For Fall 2019 ps3, we notice two high percentages of 14.29% for false negatives. These are students that did not use one of the expected

tags. This suggests that the expected tags for those expected answers might not be comprehensive enough. While the numbers for false positives are generally low, the positive percentages are all above 50%, and this also suggests that the tags are not doing a good enough job at identifying the right content.

	problem set:	ps4	ps4	ps4	ps4	ps4	ps4
	file:	ex1/Board.java	ex1/Board.java	ex2/Board.java	ex2/Board.java	ex3/Board.java	ex3/Board.java
	line range:	47-49	61	33-36	47-49	19-26	80-82
	expected answertopic:	busy	monitor	tsa	busy	rep	busy
	expected tags:	-sfb	-sfb	-sfb	-sfb	-sfb -rfc	-sfb
	# students with right content	16	15	21	18	11	18
	# students with wrong content	0	2	2	3	4	0
true positive	**right content exact tags within range**	13	5	18	11	1	12
		81.25%	33.33%	85.71%	61.11%	9.09%	66.67%
true positive	**right content missed tags within range**	1	0	0	0	3	2
		4.00%	n/a	n/a	n/a	18.18%	11.11%
true positive	**right content added tag(s) within range**	2	4	2	4	0	4
		12.50%	26.67%	9.52%	22.22%	0.00%	22.22%
false negative	**right content no right tag(s) within range**	1	1	0	3	0	2
		6.25%	6.67%	0.00%	16.67%	0.00%	11.11%
false negative	**right content wrong line**	0	8	1	0	8	0
		0.00%	53.33%	4.76%	0.00%	72.73%	0.00%
false positive	**wrong content one of right tags within range**	0	0	1	1	2	0
		n/a	0.00%	50.00%	33.33%	50.00%	n/a

Figure 5-7: ps4 practice task accuracy - fa19

For ps4, while the numbers for false positives are better, there are more issues with false negatives. We can see some pretty high percentages for the `monitor` and `rep` expected answers, with 53.33% and 72.73% of students with the right content receiving negative feedback. In contrast to ps3, we have more cases of students making comments with the right content outside of the expected line ranges. This suggests that there is a need for a more robust expected lines verification.

5.4.2 Spring 2020

As discussed in section 5.2, based on analysis of practice task feedback accuracy from ps3 and ps4 of Fall 2019, we made changes to the system for the Spring 2020 semester.

In particular, the expected line ranges for comments can be many disjoint ranges. In addition, keywords can be any string, beyond just `+/- sfb/etu/rfc` tags.

Below is the data on practice task feedback accuracy for all five problem sets of Spring 2020. The format is the same as the Fall 2019 data, except we look at keywords now. This means that there are only two cases for true positives—students with comments that use all of the expected keywords, or students that use some of them.

	problem set:	ps0	ps0	ps0	ps0
	file:	TurtleSoup.java	TurtleSoup.java	TurtleSoup.java	TurtleSoup.java
	line ranges:	23-30	57,64	58	43-44
	expected answer topic:	DRY	magicNumber	final	varName
	expected keywords:	-sfb,-rfc,DRY,repeat,loop	-etu,-sfb,magic	-sfb,final	-etu,variable name
	# students with right content	196	35	7	111
	# students with wrong content	0	3	0	0
true positive	**right content exact keywords within range**	8	16	6	55
		4.08%	45.71%	85.71%	49.55%
true positive	**right content missed some keywords within range**	171	18	1	55
		87.24%	51.43%	14.29%	49.55%
false negative	**right content no right keywords within range**	2	2	0	0
		1.02%	5.71%	0.00%	0.00%
false negative	**right content wrong line**	16	1	0	1
		8.16%	2.86%	0.00%	0.90%
false positive	**wrong content one of right keywords within range**	0	2	0	0
		0.00%	66.67%	0.00%	0.00%

Figure 5-8: ps0 practice task accuracy - sp20

The accuracy for the one ps0 practice task was pretty successful. The `DRY` expected answer had an unusually high number of students with comments with the right content but on the wrong line. In this case, students had their comment starting on the method signature, instead of inside the function body where the problem with the code was. Even though this applied to 16 students, the percentage is 8.16% since the number of students who had comments with the right content was very high—196 total students.

	problem set:	ps1	ps1	ps1	ps1	ps1	ps1
	file:	task1/Deduplicate.java	task1/Deduplicate.java	task2/BucketSetsTest.java	task2/BucketSetsTest.java	task3/BucketSetsTest.java	task3/BucketSetsTest.java
	line ranges:	29	31-33	19	25	26-28	23
	expected answer topic:	strengthen	canonical	incomplete	overlap	testCase1	testCase2
	expected keywords:	["-etu", "determin", "specific", "vague"]	["-etu", "canonical", "determin"]	["-sfb", "boundary", "incomplete"]	["-sfb", "overlap", "disjoint"]	["-sfb", "test case"]	["-sfb", "test case"]
	# students with right content	28	23	34	13	12	6
	# students with wrong content	3	5	5	4	5	13
true positive	**right content exact keywords within range**	0	1	0	0	0	0
		0.00%	4.35%	0.00%	0.00%	0.00%	0.00%
true positive	**right content missed some keywords within range**	26	21	33	13	9	4
		92.86%	91.30%	97.06%	100.00%	75.00%	66.67%
false negative	**right content no right keywords within range**	2	1	1	0	3	1
		7.14%	4.35%	2.94%	0.00%	25.00%	16.67%
false negative	**right content wrong line**	0	0	0	0	0	1
		0.00%	0.00%	0.00%	0.00%	0.00%	16.67%
false positive	**wrong content one of right keywords within range**	3	5	0	3	1	5
		100.00%	100.00%	0.00%	75.00%	20.00%	38.46%

Figure 5-9: ps1 practice task accuracy - sp20

The three ps1 practice task files also had pretty high accuracy. There are a few false negatives with high percentages for the two expected answers in the `task3/RepListIntervalS` file (last two columns), but the numbers themselves are low. In addition, this problem set introduces the idea of testing and partitions, which is what `task3` focuses on. This is a topic that students historically struggle with, especially when they first learn about it.

	problem set:	ps2	ps2	ps2	ps2	ps2	ps2	ps2
	file:	task1/IntervalSetTest.java	task1/IntervalSetTest.java	task2/IntervalSetTest.java	task2/IntervalSetTest.java	task3/RepMapIntervalSet.java	task4/RepListIntervalSet.java	task4/RepListIntervalSet.java
	line ranges:	26-27	29-32	26-27	29-32	31-32	27	28-29
	expected answer topic:	invalidBoundary	incomplete	invalidBoundary	incomplete	badSRE	invalidRI	abstractRI
	expected keywords:	["-etu", "boundary", "boundaries"]	["-etu", "incomplete"]	["-etu", "boundary", "boundaries"]	["-etu", "incomplete"]	["-etu", "-sfb"]	["-etu", "abstract"]	["-etu", "abstract"]
	# students with right content	8	10	16	13	39	7	8
	# students with wrong content	5	5	6	13	1	4	1
true positive	**right content exact keywords within range**	0	0	0	1	5	0	1
		0.00%	0.00%	0.00%	7.69%	12.82%	0.00%	12.50%
true positive	**right content missed some keywords within range**	2	3	3	6	32	4	8
		25.00%	30.00%	18.75%	46.15%	82.05%	57.14%	100.00%
false negative	**right content no right keywords within range**	6	7	14	6	0	3	0
		75.00%	70.00%	87.50%	46.15%	0.00%	42.86%	0.00%
false negative	**right content wrong line**	0	0	0	0	2	0	0
		0.00%	0.00%	0.00%	0.00%	5.13%	0.00%	0.00%
false positive	**wrong content one of right keywords within range**	1	2	3	4	0	2	1
		20.00%	40.00%	50.00%	30.77%	0.00%	50.00%	100.00%

Figure 5-10: ps2 practice task accuracy - sp20

Accuracy for ps2 practice task feedback was not as great for the first two files, with two expected answers per file (first four columns). Like `task3` of ps1 practice files, these two files are also using testing files, with the topics focused on testing partitions. The first two files contain the same two expected answers, and we wanted to see if the presence of actual JUnit test methods in a test file made a difference. In this case, the `task1` file includes JUnit tests while the `task2` file does not. We can see that fewer students wrote comments with the right content for `task1` compared to `task2`. In fact, for `task1`, each student made an average of 0.6 comments in the expected line ranges of the two expected answers. On the other hand, for `task2`, each student made an average of 0.77 comments. In conclusion, it seems like the less "distractor" code there is in the practice task file, the more likely students are to make comments of the right content in the right range.

Examining the false negative comments one by one show that there are more obvious keywords that we did not realize would be better for `task1` and `task2`.

	line ranges:	17-24	60-63	67-71	63-66	70-74
	expected answer topic:	datatypeDef	sizeSpec	flipSpec	sizeSpec	flipSpec
	expected keywords:	["-etu", "type", "datatype def"]	["-etu", "width", "height", "dimension"]	["-etu", "new expression", "return"]	["-etu", "width", "height", "dimension"]	["-etu", "new expression", "return"]
	# students with right content	49	17	11	20	14
	# students with wrong content	7	2	0	3	0
true positive	**right content exact keywords within range**	10	1	1	1	0
		20.41%	5.88%	9.09%	5.00%	0.00%
true positive	**right content missed some keywords within range**	38	14	10	2	14
		77.55%	82.35%	90.91%	10.00%	100.00%
false negative	**right content no right keywords within range**	0	1	0	0	0
		0.00%	5.88%	0.00%	0.00%	0.00%
false negative	**right content wrong line**	1	1	0	1	0
		2.04%	5.88%	0.00%	5.00%	0.00%
false positive	**wrong content one of right keywords within range**	3	0	0	0	0
		42.86%	0.00%	0.00%	0.00%	0.00%

Figure 5-11: ps3 practice task accuracy - sp20

1		problem set:	ps4	ps4	ps4
2		file:	task1/Board.java	task2/Board.java	task3/Board.java
3		line ranges:	34-36,78-79	33-36	19-26,32-37,42,44,46,48
4		expected answer topic:	monitor	fieldsTSA	badRep
5		expected keywords:	["-sfb", "monitor", "synchroniz", "lock"]	["-sfb", "relationship", "interleav", "race", "interact"]	["-sfb", "-rfc", "string", "enum"]
6					
7		**# students with right content**	19	14	14
8		**# students with wrong content**	6	5	2
9					
21	true positive	**right content exact keywords within range**	1	0	1
22			5.26%	0.00%	7.14%
23	true positive	**right content missed some keywords within range**	18	13	12
24			94.74%	92.86%	85.71%
25					
26	false negative	**right content no right keywords within range**	0	1	0
27			0.00%	7.14%	0.00%
28	false negative	**right content wrong line**	0	0	2
29			0.00%	0.00%	14.29%
30					
31	false positive	**wrong content one of right keywords within range**	3	1	0
32			50.00%	20.00%	0.00%

Figure 5-12: ps4 practice task accuracy - sp20

Problem set 3 and 4 were both pretty successful in minimizing false negatives. ps4 had a higher percentage of 14.29% for one of the false negative cases in the last column. Looking at the expected line ranges for that expected answer, we see that it is really complex. This might be a case where the actual lines on the practice task file should be rearranged for less complexity.

5.5 Additional Practice Task Experiments

5.5.1 Effectiveness of Keywords

One of the changes we made from Fall 2019 to Spring 2020 was using any string as a keyword, instead of just `+/- sfb/etu/rfc` tags. We wanted to analyze practice task comments to see how useful keywords that were not tags ended up being. To do this, we examined the keywords used in all comments that were given positive feedback—this includes true positives and false positives. The figures below show the results of this analysis. The top row of each figure is the name/topic of the

expected answer, the same topic in the figures of section 5.4. Each expected answer has a "keyword" and "count" column. The count value corresponds to the number of comments that used the keyword value to its left. Each figure below displays this information for true positives and false positives.

topic:	DRY		magicNumber		final		varName	
ps0	keyword	count	keyword	count	keyword	count	keyword	count
true positive	-sfb	135	-etu	29	-sfb	6	-etu	109
	-rfc	141	-sfb	16	final	7	variable name	56
	DRY	62	magic	31				
	repeat	44						
	loop	159						
false positive	-sfb	0	-etu	0	-sfb	0	-etu	0
	-rfc	0	-sfb	0	final	0	variable name	0
	DRY	0	magic	2				
	repeat	0						
	loop	0						

Figure 5-13: ps0 keyword usage - sp20

For ps0, we notice that the keywords that are not tags are pretty effective for each expected answer. In fact, for every expected answer except for the last one (`varName`), the most used keyword is not a tag.

topic:	strengthen		canonical		incomplete		overlap		testCase1		testCase2	
ps1	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count
true positive	-etu	26	-etu	16	-sfb	33	-sfb	13	-sfb	9	-sfb	4
	determin	0	canonical	1	boundary	0	overlap	0	test case	0	test case	0
	specific	2	determin	12	incomplete	0	disjoint	4				
	vague	2										
false positive	-etu	3	-etu	2	-sfb	0	-sfb	3	-sfb	1	-sfb	5
	determin	0	canonical	2	boundary	0	overlap	0	test case	0	test case	0
	specific	0	determin	1	incomplete	0	disjoint	1				
	vague	0										

Figure 5-14: ps1 keyword usage - sp20

For ps1, the keywords that are not tags are not as successful. While the `canonical` expected answer had "determin" as a pretty effective keyword, the rest of the expected answers' true positive comments relied heavily on tags. On the flip side, false positive comments are mostly a result of using tags as well.

topic:	invalidBoundary		incomplete		invalidBoundary		incomplete		badSRE		invalidRI		abstractRI	
ps2	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count
true positive	-etu	2	-etu	2	-etu	3	-etu	4	-etu	13	-etu	3	-etu	8
	boundary	0	incomplete	1	boundary	0	incomplete	4	-sfb	29	abstract	2	abstract	1
	boundaries	0			boundaries	0								
false positive	-etu	1	-etu	2	-etu	1	-etu	4	-etu	0	-etu	2	-etu	1
	boundary	0	incomplete	0	boundary	2	incomplete	0	-sfb	0	abstract	0	abstract	0
	boundaries	0			boundaries	0								

Figure 5-15: ps2 keyword usage - sp20

For ps2, `incomplete` and `invalidRI` have successful use of keywords that were not tags. For false positives, the comments mostly used tags, simlar to ps1, with the exception of "boundary" keyword.

topic:	datatypeDef		sizeSpec		flipSpec		sizeSpec		flipSpec	
ps3	keyword	count	keyword	count	keyword	count	keyword	count	keyword	count
true positive	-etu	45	-etu	14	-etu	10	-etu	2	-etu	12
	type	48	width	4	new expression	2	width	1	new expression	1
	datatype def	10	height	4	return	7	height	1	return	11
			dimension	5			dimension	0		
false positive	-etu	1	-etu	0	-etu	0	-etu	0	-etu	0
	type	2	width	0	new expression	0	width	0	new expression	0
	datatype def	2	height	0	return	0	height	0	return	0
			dimension	0			dimension	0		

Figure 5-16: ps3 keyword usage - sp20

ps3 practice tasks are an example of more successful use of keywords beyond tags, especially the `datatype_def` expected answer. In addition, false positives are zero for all expected answers except the first. In that case, the keywords students used for false positive comments are mostly non-tags.

topic:	monitor		fieldsTSA		badRep	
ps4	keyword	count	keyword	count	keyword	count
true positive	-sfb	19	-sfb	12	-sfb	10
	monitor	10	relationship	0	-rfc	6
	synchroniz	12	interleav	1	string	4
	lock	5	race	2	enum	7
			interact	1		
false positive	-sfb	3	-sfb	1	-sfb	0
	monitor	2	relationship	0	-rfc	0
	synchroniz	1	interleav	0	string	0
	lock	0	race	0	enum	0
			interact	0		

Figure 5-17: ps4 keyword usage - sp20

The first and third expected answer for ps4 show good use of keywords that are not tags, while the second expected answer's true positive comments use mostly the

`-sfb` tag. For false positives, the first expected answer has half of its false positive comments use keywords that are not tags ("monitor" and "synchroniz").

In order to investigate false positive comments even deeper, we looked at the specific comments that are false positives and use a keyword that is not a tag. In these cases, the student commented on the right topic, but the content of the comment was the opposite of what we expected. For example, for the ps3 `datatype_def` expected answer, the comments that use "type" and "datatype def" keywords write that the datatype definition in the practice task file was good. However, the datatype definition is actually flawed, and the right comment to make is to explain that it is wrong and give a reason why.

5.5.2 Spring 2020 Problem Set 3 Experiment

The last experiment we ran for the practice task system aims to see if the topics of expected answers would affect the content of comments made in the regular code review tasks that came after the practice task. Problem set 3 has two practice task files, both named `Expression.java`. These files are the same, except the first one has a flawed datatype definition.

The way that problem set 3 is designed, every student has to have an `Expression.java` file in their own problem set code, and all of these files accomplish the same task of defining an interface that is required for this assignment.

We looked at all comments made on `Expression.java` files for regular code review tasks—these are comments made by students on other students' files. For students who had the `task1` practice file with an expected answer about a bad datatype definition, we hope to see that they made similar comments about datatype definitions on other students' code. The way that we try to measure this effect is to look at the average number of comments and characters used by students on future `Expression.java` files.

For students who received `task1` (with a *bad* datatype definition), they made an average of 2.2 comments and wrote an average of 184.6 characters on future `Expression.java` files.

For students who received `task2` (with a *good* datatype definition), they made an average of 2.4 comments and wrote an average of 229.8 characters on future `Expression.java` files.

Students with the practice task file with a good example of a datatype definition actually wrote more comments and more characters on future `Expression.java` files. This suggests that a good example may stick with students more than a bad example with an expected answer. The numbers may also be too close to come to a conclusion. Another hypothesis is that students do not really have bad datatype definitions for other students to comment on.

To explore this more, we looked through all the student comments on other student's `Expression.java` files. We found two comments that are related to the isuue in the `datatype_def` expected answer of `task1`. The first comment was by a student who received `task1` as a practice task. The comment said "good job" to the code author for writing the datatype definition correctly, specifically giving praise for avoiding the datatype definition issue from the practice task. The second comment was by a student who received `task2`. Even though `task2` has a great datatype definition, this comment pointed out the exact datatype definition issue from the `task1` practice task. From these findings, we see that a good and bad example can influence students to write comments on the same topic in regular code reviews.

To build off of this, we performed the same analysis for the two other expected answer topics. These expected answers are the same for both tasks, with names of `flipSpec` and `sizeSpec`. Looking at student comments on other students' `Expression.java` files, we found 9 comments about the same problem from the `flipSpec` expected answer, and 6 comments about the same problem from the `sizeSpec` expected answer. This is strong evidence that seeing a problem from a practice task file can result in students commenting on that same problem if it appears in regular code review tasks.

Chapter 6

Discussion

6.1 `sfb/etu/rfc` Tags

In the evaluation, we looked at comment quantity in order to see how the addition of tags influenced code review in the Fall 2019 semester and onward.

Looking at data on the number of characters per comment and the number of comments per student per task, there is not strong evidence that having tags in the Fall 2019 and Spring 2020 semester resulted in higher values for those semesters compared to the Spring 2019 semester.

However, we believe there needs to be more evaluation done to determine the effects of tags in Caesar. Future evaluation might include A/B testing, where half of student have a code review interface that uses tags for comments, and the other half does not. Then, it might be easier to compare how quantity and quality of comments made by those two groups differ.

6.2 Practice Task Feedback Accuracy

A bulk of evaluation done for this thesis includes evaluating the accuracy of automated feedback comments made by the practice task system. In particular, we want to minimize false negatives and false positives, to make sure students receive positive feedback for comments of the right content.

Overall, the changes made between the Fall 2019 and Spring 2020 result in lower false negatives in the Spring 2020 practice tasks. Specifically, we decreased the number of false positives due to comments not being in the expected line range—this is the result of allowing multiple, disjoint expected line ranges.

From the analysis of keywords in section 5.5.1, we see that there were expected answers for which the usage of keywords beyond tags was very useful. However, there are also many expected answers for which this was not the case. More work should be done on this part of the system to develop a more accurate and robust feedback system. Developing a more accurate way to provide feedback based on keywords used would also decrease the rate of false positives, which did not see a significant decrease in Spring 2020. Since false positives result from a student writing a comment using at least one expected keyword, in the right range, but with the wrong content, future work should focus on how to prevent that case.

6.3 Effect of Practice Tasks on Regular Code Review Tasks

We looked at comment quality and did some other practice task experiments to see what sort of effect practice tasks had on comments made on regular code review tasks.

When looking at data about comment quality (LA grading of student comments), we did not notice that either Fall 2019 or Spring 2020 had better comments consistently. As mentioned in section 5.1, the Spring 2020 semester had special circumstances that we believe skew the data collected. In addition, since LA grading of comments was newly introduced this past school year, we are not sure how consistent each LA may be with other LAs. As a result, we think that more evaluation on comment quality should be done to see how practice tasks might affect quality. Similar to comment quantity, A/B testing could be useful here. In addition, comparing grades given by the same LA across different problem sets or A/B testing groups could provide more insight without the noise from grading inconsistency across different

LAs.

Finally, in section 5.5.2, we performed more detailed analysis on comments from ps3 code review in the Spring 2020 semester. In this analysis, we saw that the expected answer topics in practice tasks do influence the kinds of comments students make on regular code review tasks after they have completed the practice task. This analysis also shows that there is value in having expected answers for practice tasks be code issues that students actually make in their own code. This is something that we already think about in the creation of practice problems and their expected answers, and our analysis reinforces that idea.

Chapter 7

Future Work

7.1 Keywords Accuracy

To further reduce false positives and false negatives, the current usage of keywords in giving automated feedback can be improved. Currently, receiving positive feedback requires a comment to be in the right location and use at least one expected keyword. Depending on the choice of keywords, false positives may happen easily if the keyword is not specific enough for the expected answer. In particular, we noticed that this happens more often for keywords that are `+/- sfb/etu/rfc` tags. However, we believe that having these tags are useful, since they draw focus to the three principles of 6.031. In addition, being able to mark a comment with a + or − is useful.

In order to improve how keywords are used, future work could "tie" certain keywords together. For example, if the set of keywords for an expected answer is `["-etu", "type", "datatype def"]`, then perhaps the system would only give positive feedback if "etu" is used with one of "type" or "datatype def". This way, false positives that result from a student writing a comment that only uses "etu" would be avoided. This change could increase the chance of false negatives, since it would be harder to get positive feedback, so testing would definitely be required. This type of idea can also be applied to some expected answers and not others. For example, expected answers that have very obvious non-tag keywords could benefit from this change.

Another future change could be adding a post code review processing script that helps with the selection of keywords. After code review concludes, this script would analyze all the comments made for practice tasks and suggest new keywords that could be used.

7.2 Keyword Robustness

The current system utilizes keywords by looking for a direct match in the text of practice task comments. Future work could expand this beyond direct matches, such as using a regex matcher instead, allowing functionality like keywords representing suffixes or prefixes.

Chapter 8

Conclusion

This thesis presents three changes to the Caesar system—`+/-` `sfb/etu/rfc` tags, a LA grading interface for student comments, and practice tasks. These changes are designed to help Caesar be a more useful code review system in the classroom, by making it easier for novice student code reviewers to provide high quality code review comments.

The introduction of tags to writing new comments create more structure for students to follow when writing comments. Since our evaluation did not show a strong effect from this change, future analysis is needed. At minimum, we believe that adding structure to push students to think about the three core principles of 6.031 when writing comments is a positive goal.

The LA grading interface allows the staff to collect more information on the quality of comments written by students. It also simplifies how 6.031 grades student participation in code review. Since this is a new interface for LAs, we hope to keep it for future semesters. We also hope that future data from LA grading will allow us to evaluate the practice task system and the tags in comments even more.

Finally, the introduction of practice tasks provides an initial training period for student code reviewers. We improved the system over the past school year to provide more accurate feedback, and we have seen that practice tasks result in students learning from their content and looking out for similar problems in other students' problem set code. There is more work to be done in improving feedback accuracy,

but the main contribution of this thesis is providing a working practice task system that aims to improve student code review quality.

The results shown in this thesis support that crowdsourcing techniques applied to code review in the classroom can be effective. We hope to continue applying and refining such techniques for 6.031, ultimately preparing students to be better code reviewers and collaborative software engineers.

Bibliography

[1] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, and H. R. M. Nezhad. Quality control in crowdsourcing systems: Issues and directions. *IEEE Internet Computing*, 17(2):76–81, March 2013.

[2] M. D. Greenberg, M. W. Easterday, , and E. M. Gerber. Critiki: A scaffolded approach to gathering design feedback from paid crowdworkers. *2015 ACM SIGCHI Conference on Creativity and Cognition*, pages 235–244, June 2015.

[3] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. *2008 Conference on Human Factors in Computing Systems*, April 2008.

[4] Abigail Klein. Search tools for scaling expert code review to the global classroom. Master's project, Massachusetts Institute of Technology, September 2015.

[5] C. Kulkarni, K. P. Wei, H. Le, D. Chia, K. Papadopoulos, J. Cheng, D. Koller, , and S. R. Klemmer. Peer and self assessment in massive online classes. *ACM Transactions on Computer-Human Interaction*, 20(6):33, December 2013.

[6] J. Le, A. Edmonds, V. Hester, and L. Biewald. Ensuring quality in crowdsourced search relevance evaluation: The effects of training question distribution. *SIGIR 2010 Workshop on Crowdsourcing for Search Evaluation*, July 2010.

[7] Mason Tang. Caesar: A social code review tool for programming education. Master's project, Massachusetts Institute of Technology, September 2011.

[8] Elena Tatarchecko. Analysis of performing code review in the classroom. Master's project, Massachusetts Institute of Technology, June 2012.

[9] A. Yuan, K. Luther, M. Krause, S. Vennix, S. P. Dow, and B. Hartmann. Almost an expert: The effects of rubrics and expertise on perceived value of crowdsourced design critiques. *19th ACM Conference on Computer-Supported Cooperative Work Social Computing*, pages 1005–1017, February 2016.