Analysis of Performing Code Review in the Classroom

by

Elena Tatarchenko

S.B., Massachusetts Institute of Technology (2011)

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 and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

© Massachusetts Institute of Technology 2012. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified by ....
Robert C. Miller
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Analysis of Performing Code Review in the Classroom

by

Elena Tatarchenko

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical and Computer Science

Abstract

In real-world software development, engineering teams generally engage in code review to ensure the authorship of high-quality code. However, in undergraduate university settings, where many students are preparing for a career in software development, code that is written for classes generally does not go through similar code review processes. There are many aspects of the academic setting which does not allow existing commercial code review processes to be used. This work presents a software solution to code reviewing in academic settings, so that students can get constructive comments about the code they turn in for assignments. The software was used in two semesters of the Software Engineering course at MIT, and experimental results show that students can generate high-quality feedback for code written by their classmates.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, Rob Miller, who provided much advice, guidance, and insight throughout my research. He allowed me to explore my own ideas while still keeping me on track. He constantly inspired me to work harder and to come up with new and innovative solutions.

A special thanks goes to Mason Tang who started this project. He had a vision of code review being used in the classroom and that inspired my work on this thesis. He passed on to me a functional, well written system that I could apply to 6.005 and run experiments.

I would also like to thank both the Fall and Spring 6.005 staff for allowing this tool to be used and being patient with it when it did not perform to its full potential. As users of the system you also offered me feedback that was especially valuable. Thank you for making the tool a success.

Finally I would like to thank my friends and family for being supportive when I was working on this project. Thank you for tirelessly listening to my problems and brainstorming solutions. Some of you went well beyond your duties and helped me debug code, participate in code review, and even edit this thesis.

Contents

1 Introduction **13**

1.1 Mechanics of Assigning Code to Review 14

1.2 Experiment Setup 15

1.2.1 Chunk Size 15

1.2.2 Chunk Filtering 16

1.2.3 Task Routing Interface 17

1.3 Outline 17

2 Code Review in 6.005 **21**

2.1 The Interface 21

2.2 Workflow 23

2.3 6.005 24

2.4 Participants of Code Review 26

3 Related Work **29**

3.1 Code Review Systems 29

3.2 Defect Rates 30

3.3 Variations in Code Review 30

4 Selecting Chunk Size **33**

4.1 Motivation 34

4.2 Results 37

4.2.1 Examples 38

4.2.2 Data . . . 38
4.2.3 Potential Shortcomings . . . 41

5 Chunk Filtering **43**
5.1 Partitioning . . . 44
5.2 Preprocessing . . . 44
5.3 Changes to Display . . . 45
5.4 Routing Details . . . 45
5.5 Routing Based on Branching Depth . . . 50
5.6 Shortcomings . . . 51

6 Routing Interface **53**
6.1 Components of the Interface . . . 53
6.2 Responses to Interface . . . 55

7 Conclusion **57**
7.1 Future Work . . . 57
7.2 Summary . . . 59

List of Figures

1-1 Routing interface helping staff calculate how much code will be reviewed. . 18
1-2 Routing interface allowing staff to drag and drop chunks in order of priority. 19
2-1 Interface for seeing assigned and completed tasks. 22
2-2 Interface for leaving a comment. 22
2-3 Interface for replying to a comment. 22
2-4 Summary page displaying list of comments left by a user. 23
2-5 Feedback Survey used by Caesar. 24
2-6 Path from assignment to code review. 25
2-7 Interaction between preprocessor and web application. 26
4-1 Stacked bar plot of types of comment. 39
4-2 Bar plot of average number of comments per student. 40
5-1 Code section containing student and staff lines. 46
5-2 Bar plot showing average number of words and average number of comments left by students. 48
5-3 Bar plot of average problem set grades. 48
5-4 Bar plot of review time. 49
5-5 Fall versus spring average number of words per student. 50
5-6 Fall versus spring average number of comments per student. 51
6-1 Reviewing interface showing how many chunks can be reviewed. 54
6-2 Reviewing interface showing distribution of student lines. 55
6-3 Reviewing interface for prioritizing chunks. 56

List of Tables

2.1 Problem sets used for 6.005 in Fall 2011 and Spring 2012. 26

4.1 Statistics on comment type for each problem set. 39

5.1 Results of varying chunk sizes and new routing algorithm. 47

5.2 Results of varying chunk sizes and new routing algorithm. 49

Chapter 1

Introduction

Code review is a process where computer code is inspected by other programmers, with the intention of finding bugs and improving code readability. Code review is necessary in industry to ensure maintainable code, but has not had widespread adoption in an educational setting. In a large computer science class, there is generally not enough course staff to meticulously review every student's programming submission. The traditional way in which this problem is solved is to write a suite of automated tests, so that students can be graded in an efficient manner. However, this method does not expose poor design, stylistic problems, or subtle bugs; it only ensures that a student's code is generally correct.

Prior work done by Mason Tang [15] produced a software system called Caesar. Caesar is a software suite designed to solve the problem of code review in the classroom. The innovation in Caesar is that the code review problem is distributed to both the students and staff of the course. A part of the grade for any problem set is that students have to both give reviews for other peoples' code, as well as address the reviews on their own code. This crowdsourced solution solves the problem of giving students constructive feedback, without having to increase the size of the course staff. The work by Tang produced the foundation of Caesar, in that the basic pipeline of getting student code into a database was written, and a basic web application that exposed student code to other students was built.

This work is a refinement on top of the existing Caesar project. The ultimate goal

of a system like Caesar is to maximize the amount of constructive feedback that any student gets on their problem set submissions. It is an unexplored problem as to how to best divide student code to optimize it for code reviewing purposes, as well as how to distribute those code chunks to potential reviewers. The primary contribution of this work is the development of algorithms and evaluations to try to optimize how student and staff time is spent, while producing the most relevant feedback for all submitted student code.

In addition, the project had never actually been used in a real classroom setting before; this work partially describes the process of launching Caesar in two semesters of a software engineering class at MIT. The algorithms for dividing student code were developed on top of real student submissions in these two semesters, and data produced by students using the system is the foundation of the evaluations presented in this work.

1.1 Mechanics of Assigning Code to Review

A critical component of the system is deciding how much and what code should be given to each reviewer to review. In industry, code typically lives in a repository, and each code modification is called a commit. Before a commit can occur, a code review happens on that commit[10].

However, this is not the paradigm that is generally used in classroom settings. Even though a source control system is provided to students, students generally don't have the discipline to use it in an effective manner. In addition, student commits shouldn't necessarily be graded anyways, since they are merely using it as a way to store history, and not as a method of submitting working code at each revision. Students turn in one final submission, which is a monolithic set of interdependent files. A single assignment may be thousands of lines of code, some of which are staff provided.

If the project is too large, it is infeasible to assign that project to just one reviewer; it would be too much work for one person to do. However, instead of assigning a

reviewer to an entire submission, we can partition each submission, and group similar segments of code for each reviewer. Each code segment that Caesar treats as a discrete unit will be referred to as a *chunk*. A reviewer may be assigned several chunks during the reviewing process, and each chunk could be assigned to several reviewers. A chunk that has been assigned to a reviewer will be referred to as a *task*. It is the aim of this project to figure out what is the best way to create, and then distribute chunks.

1.2 Experiment Setup

Caesar was launched for the Fall 2011 semester of Elements of Software Construction (6.005) class at MIT, and was used through Spring 2012. The class was taught entirely in Java. During that time, the system was used for a total of 13 problem sets. Five of the problem sets that were used in Fall 2011 were also used in Spring 2012. As the algorithms governing chunk size and routing changed, experiments can be constructed that observe what effect algorithm changes had on the number of comments that were entered into the system.

Because the project is trying to maximize the number of *constructive* comments that are left to students, part of the evaluation process involves observing whether or not students truly left constructive comments, or if they were not useful. Although there is no objective way classify comments, a random set of comments was chosen to be classified, and observations are made based on how many provided useful input to the code author.

1.2.1 Chunk Size

Two different methods for creating chunks were created, and both were used to break up real student code. The first idea that was implemented was that a chunk should be a single Java method. This way, the average size of a chunk is small, but there would be more chunks assigned to each reviewer. The second idea is that chunks should be entire Java classes. Each reviewer would get fewer chunks to review, but each would be more substantial.

This work evaluated the merits of each chunking algorithm by comparing the type of comments that were entered in the system in Fall 2011. Since the goal of the project is to maximize the number of constructive comments, we evaluate the success of the system by comparing the total number of constructive comments entered in the system under these two settings. The results of this experiment showed that more constructive comments were entered when chunks were set to be classes.

1.2.2 Chunk Filtering

In most cases, there are not enough reviewers for the amount of code that is generated, without giving each reviewer an unfeasible amount of code to review each week. In addition, a lot of code that students submit is either provided by the staff as part of the assignment, or other boilerplate code that is not necessarily important to review. Another challenge of the system is picking out which chunks need the most attention from reviewers.

A component of Caesar is an algorithm to rank chunks for review that prioritizes the most needy code for review, while not prioritizing uninteresting code. Ideally, if all the code was reviewed, the chunks that would have received the most comments would be the ones that the system originally prioritized for reviewing. There are a myriad number of signals that can be used from student code which could serve as an indicator of whether or not that code needs to be reviewed. One signal that was found to be useful is *branching depth*, the amount of conditional statements used in a snippet of code.

To evaluate changes to the chunk routing algorithm, a natural experiment was conducted when the same problem sets were used between Fall and Spring of the same course. Chunk size was kept constant, and a comparison was made between the number of comments made between the two semesters. The results of the experiment showed that prioritizing chunks based on the number of student lines and branching depth was more effective than prioritizing strictly based on number of student lines.

1.2.3 Task Routing Interface

Another problem this work tries to address is how to give the staff more control over what code will be reviewed. Each problem set is inherently different, and both the chunk size and chunk filtering algorithms should be adaptive to the needs of the course staff for any given assignment.

To solve this problem, an interface was constructed for visualizing the code in the system, and for fine tuning the reviewing process for that particular problem set. Figures 1-1 and 1-2 show the components of this interface. The interface helps staff understand how many chunks of code will be reviewed. It also lets staff pick which chunks to select for review, and how they should be prioritized. This adds flexibility to the system, and lets staff guide the reviewing process for each problem set.

1.3 Outline

The remainder of this work explores the algorithms that were developed for Caesar, as well as evaluations of those algorithms using real data. Chapter 2 describes the individual components of Caesar, and how a workflow generally operates in the context of a course problem set. Chapter 3 explores the related work in code review, as well as ways that Caesar could be used to tackle other previously mentioned problems. Chapter 4 describes the algorithm and evaluation for choosing chunk size. Chapter 5 talks about the algorithms and evaluations for filtering and ranking chunks. Chapter 6 shows the interface built for staff that lets them control and tune the chunking and filtering algorithms. Finally, chapter 7 contains future work that can be done on Caesar, as well a summary of the findings of this work.

Dashboard | Assignment Info

view all users | elena_11 (108) | Admin | Summary | Manage | Logout

Settings for ps2-beta.

Assign [2] non staff users (students and alums) to each class for review. Staff will serve as quality control and be assigned as a 3rd reviewer to as many classes as possible.

For the reviewing process we can expect:

Baseline values for ps2-beta in *purple*.

199 *199*	students	reviewing	5 *5*	tasks each.	199 x 5 = 995 tasks total.
1 *1*	alums	reviewing	3 *3*	tasks each.	1 x 3 = 3 tasks total.
15 *15*	staff	reviewing	10 *10*	tasks each.	
					/ (2 tasks/class) = 499 classes

At most 499 classes will reviewed.

Only classes with at least [30] student lines will be considered for review.

Click and drag in the plot area to zoom in.

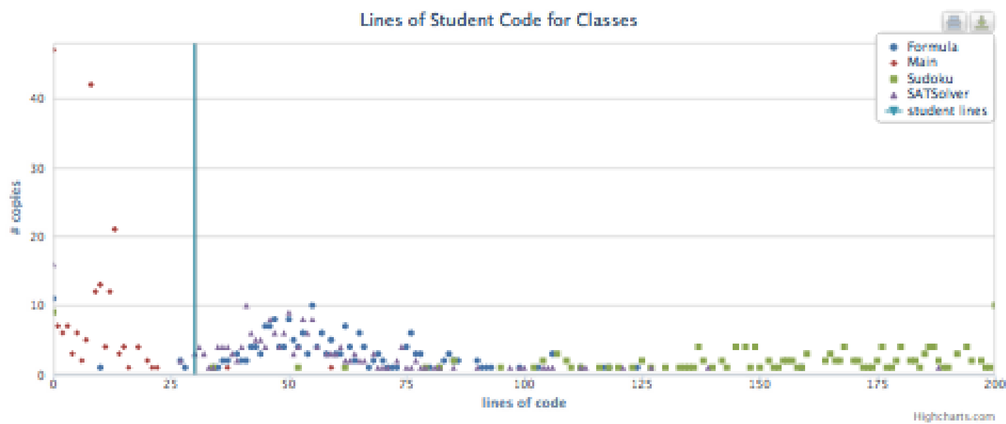

Figure 1-1: Routing interface helping staff calculate how much code will be reviewed.

Check which classes should be reviewed
Drag to prioritize reviewing order.

- [x] Formula, 190
- [x] Main, 2
- [x] Sudoku, 146
- [x] SATSolver, 187
- [ ] SATSolverTest, 170
- [ ] FormulaTest, 164
- [ ] SudokuTest, 156

The following classes are too short
(in terms of student lines):

Environment
Literal
Bool
Empty
PosLiteral
AssocList
Variable
ImList
ImListIterator

Figure 1-2: Routing interface allowing staff to drag and drop chunks in order of priority.

Chapter 2

Code Review in 6.005

2.1 The Interface

Caesar is an application developed at MIT that solves the problem of code review in the classroom. The project was started by Mason Tang in Spring 2011 [15], but not used by students until Fall 2011. He designed the interface for displaying code, leaving comments on source code, and displaying a dashboard for announcements and tasks.

The dashboard is the first page users see when they log onto Caesar. Their dashboard contains a list of tasks they are asked to complete and a list of tasks that they have already completed. Figure 2-1 shows an example of the dashboard the user sees. Each line shows information about the code, such as how many reviewers are assigned to it, and the total number of comments that have already been left.

The interface for leaving comments is lightweight and easy to use. To leave a comment, users click and drag over several lines of source code. When they release the button, a text field next to their highlighted region appears. Figure 2-2 shows this in action. Users may also reply to existing comments by clicking on the "reply" button when they are hovering over the comment, as in Figure 2-3. The comment bubbles next to the source code are clickable, and highlight the comment and source code that the comment bubble represents.

In addition, there is summary page for each user that displays a list of comments

Dashboard

view all users | elena_11 (108) | Admin | Summary | Manage | Logout

Welcome to Caesar, the 6.005 code reviewing system. Please read the code reviewing instructions on Stellar if you haven't already.

The problem set being reviewed now is Problem Set 4: Jotto Client GUI The gray lines who student-authored code; white lines were provided by staff.

code to review

JottoModel	package model; import java.io.BufferedReader; import java.io.IOException; import j...	3	2
JottoModel	**package model; import java.io.BufferedReader; import java.io.IOException; import j...**	**5**	**2**
JottoModel	**package model; import java.io.BufferedReader; import java.io.IOException; import j...**	**8**	**2**

code already reviewed

RulesOf6005	import static org.junit.Assert.assertEquals; import java.util.ArrayList; import j...	12	3
RulesOf6005	import java.util.Calendar; import java.util.GregorianCalendar; /** * RulesOf6005 r...	8	3
RulesOf6005	import java.util.Calendar; import java.util.GregorianCalendar; import java.lang.Ma...	8	3
RulesOf6005	import java.util.Calendar; import java.util.GregorianCalendar; import java.lang.Ma...	6	3

Figure 2-1: Interface for seeing assigned and completed tasks.

view all code

26 String[] features = new String[]...
33 weeks ago by

In java string arrays (and many other arrays involving primatives), the array can be declared and initialized with just the list between the braces. The 'new' keyword is not needed. EX: String[] oneDimArray = { "abc","def","xyz" };

0 0

29 - 34 "team meetings","quizzes" };...

Save Cancel

35 { if (name.toLowerCase().equals...
automatically generated by checkstyle

'{' should be on the previous line.

0 0

```
17 /**
18  * Tests if the string is one of the items in the Course Elements section.
19  *
20  * @param name - the element to be tested
21  * @return true if <name> appears in bold in Course Elements section. Ignores case (capitalization).
22  * Example: "Lectures" and "lectures" will both return true.
23  */
24 public static boolean hasFeature(String name){
25     //List of features from site.
26     String[] features = new String[]{
27             "lectures","recitations","laptops required",
28             "text","problem sets","code review","projects",
29             "team meetings","quizzes"
30     };
31 
32     //iterate through features and compare them to input
33     //return true if the feature and input match.
34     for ( String feature : features )
35     {
36         if ( name.toLowerCase().equals(feature) )
37             return true;
38     }
39 
40     //if the input feature is not on the list, return false.
41     return false;
42 }
```

Figure 2-2: Interface for leaving a comment.

26 String[] features = new String[]
33 weeks ago by

In java string arrays (and many other arrays involving primatives), the array can be declared and initialized with just the list between the braces. The 'new' keyword is not needed. EX: String[] oneDimArray = { "abc","def","xyz" };

0 0 Reply

Figure 2-3: Interface for replying to a comment.

the user has made for each problem set, as shown in Figure 2-4. Each user may see each other user's summary page. This view may be used by the staff to evaluate how well the students are performing code reviews, and by other students to get a better idea of the type of comments their peers and staff are leaving.

ps2 - Fall 2011 view with code

+2 You could also use DEFAULT_INSTRUMENT	PianoMachine.PianoMachine()	Sept. 26, 2011, 7:25 p.m.
Delete this print out statement since it was only used for debugging.	PianoMachineTest.InstrumentTest()	Sept. 26, 2011, 3:28 p.m.
+1 Important test to have.	PianoMachineTest.notesSameTimeTest()	Sept. 25, 2011, 1:36 p.m.
+1 Isn't this unnecessary since you don't use .showHistory()'s String for the test?	RecordingTest.recordMultipleInstrumentsOverBoundariesTest()	Sept. 23, 2011, 2:56 p.m.
+1 Isn't this unnecessary since you don't use .showHistory()'s String for the test?	RecordingTest.recordMultipleOctavesTest()	Sept. 23, 2011, 2:55 p.m.

Figure 2-4: Summary page displaying list of comments left by a user.

To encourage feedback from users, Caesar had a link to a survey that asked the users what they liked, didn't like, and any encountered problems while using Caesar. This feedback is referred to later in this work as anecdotal evidence of user preferences after algorithm changes.

2.2 Workflow

Alongside Caesar, there is preprocessor that reads in student code and starter code, then uploads partitioned code into a database. The system takes in starter code in order to mark which parts of the code are written by students, and which were provided by staff. To start out, the system processes and partitions student code into chunks. Chunk size will be discussed in more detail in Chapter 4.

The preprocessor is also responsible for catching basic stylistic mistakes. It does so by performing static analysis on the code in the form of Checkstyle[2]. Static analysis catches stylistic problems in student code such as inconsistent indenting and poor javadoc usage. This analysis is uploaded to the database and used by the web application. The preprocessor and the web application interact as shown in Figure 2-7.

Feedback Survey

Thanks for trying Caesar! We would greatly appreciate if you could tell us about your experience using our tool.

Reviewing the code
How did you go about reading and commenting on the code? Did you feel you had enough information to do a good job? Did the UI support what you wanted to do? Any problems or suggestions?

Looking at comments
Did you read and respond to other comments? Did the UI support what you wanted to do? What did you think of the automatic (CheckStyle) comments? Any problems or suggestions?

Was it interesting?
Did you find this interesting, and would you be willing to do it again to help MIT students learn how to program better? Any suggestions?

Any other thoughts?

Figure 2-5: Feedback Survey used by Caesar.

After the preprocessor is finished, the staff may need to change how many tasks to assign to a reviewer, and choose which tasks to prioritize for review. Chunk filtering will be discussed in more depth in Chapter 5 and changing the settings to the routing will be discussed in Chapter 6. Figure 2-6 shows the series of events that occur before reviewing can open.

2.3 6.005

Caesar was built for code review in the classroom, and is being used in the Elements of Software Construction (6.005) class at MIT. Each semester, there are typically four to eight individual problem sets, where each problem set consists of hundreds of lines of code. Some of the problem sets are templated, where overall design and method signatures were done by the staff. Other problems sets were more free-form, and gave students design freedom. The starter code may contain implementations of algorithms, or ask students to to come up with their own algorithms. All the code

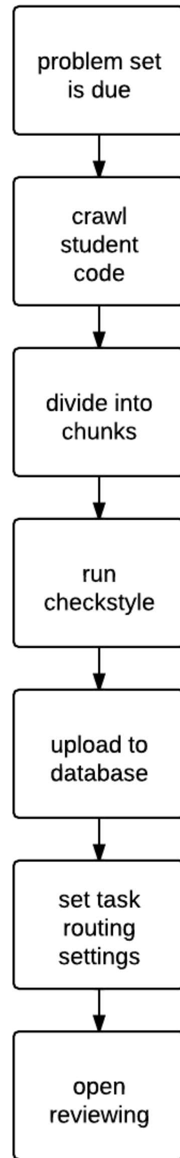

Figure 2-6: Path from assignment to code review.

is written in Java and each of the of the problem sets have automated tests that are run to verify the correctness of the code.

Fall 2011 and Spring 2011 were structured differently, but all the problem sets using in Spring were also used in the Fall. Table 2.1 shows a list of problem sets used in each semester. Fall 2011 had a total of eight problem sets, and each problem set had to be completed in a week. After the due date, the problem set would be code

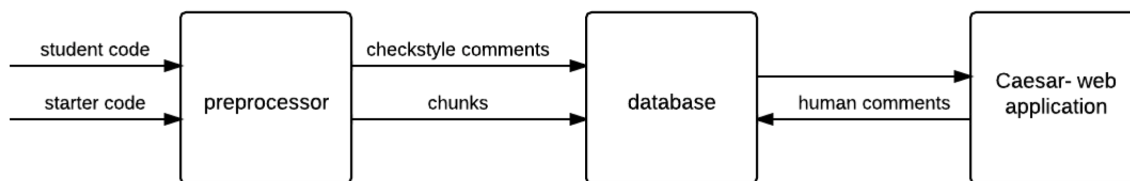

Figure 2-7: Interaction between preprocessor and web application.

reviewed using Caesar and graded using automated tests. Once students received their grades back, they had an opportunity to submit a new version of their problem set, but only if they addressed the code review comments in their new submission. Spring 2012 had a total of five problem sets, but each problem set had a beta and a final. The beta would be code reviewed and graded using the automated tests. For the final version, students were responsible for fixing their mistakes such that the automated tests would pass, as well as addressing code review comments.

Problem Set	Title	Fall 2011	Spring 2012
`intro`	Introduction	PS0	PS0
`pipoetry`	Pi Poetry	PS1	PS1
`piano`	Midi Piano	PS2	
`calculator`	Calculator Parser	PS3	
`sudoku`	Building a Sudoku Solver with SAT	PS4	PS2
`factors`	Finding Prime Factors with Networking	PS5	
`minesweeper`	Multiplayer Minesweeper	PS6	PS3
`jotto`	Jotto Client GUI	PS7	PS4

Table 2.1: Problem sets used for 6.005 in Fall 2011 and Spring 2012.

2.4 Participants of Code Review

The participants of code review in the classroom are not necessarily experts. In our system, there are three types of reviewers: students, staff, and alums. Students are the most familiar with the problem set but do not necessarily know good coding practices. Staff are both familiar with the problem set and can judge the quality of the code. Alums have taken the class at some point and in most cases have industry experience.

Our system attempts to get a diverse set of reviewers for each student's submission. The hope is that students will know where to look for challenging portions of the problem set and alums will have enough general knowledge to find design flaws.

Students and alums perform the bulk of the reviewing. However, since alums have limited time to devote to code review, they are only assigned 3 tasks, while students are assigned 10 method-sized chunks or 5 class-sized chunks. After students and alums have finished leaving their feedback, staff are assigned 20 method-sized chunks or 10 class-sized chunks that have already been reviewed by students and alums. Their responsibility is to upvote or downvote reviewer comments, and add their own comments if anything was missed.

In order to encourage reviewers to read the code they are assigned to review, the system requires reviewers to perform at least one action on their assigned chunk. The action could either be to upvote or downvote an existing comment or to leave a new comment. In the case that the reviewer could not find fault with the code, they were asked to write "#lgtm", meaning "looks good to me".

Chapter 3

Related Work

3.1 Code Review Systems

Existing industry code review tool are generally designed to interface with version control systems, support inviting teammates to participate in code review, and participate in discussion. Such systems include Rietveld [6], Gerrit [3], Github[4], and Review Board[9]. The workflow of these systems is for the developer to initiate a code review. Typically these systems ask the developer to select the code to be reviewed from the set of changes they made, and pick the reviewers they want. The tools expect users to be familiar with how code reviews work, and how to utilize them effectively. Also, these systems generally take advantage of version control techniques and display versioning comments during the code review period.

In an academic setting the life of the project is much shorter and versioning information may be nonexistent or unhelpful so our system is trying to tackle a different problem than the current review systems out there. When designing our systems we can still try to base the reviewing interface on theirs but our techniques for selecting which code to review and who should perform the code review is vastly different.

3.2 Defect Rates

According to a case study on lightweight code review process at Cisco, "the single best piece of advice we can give is to review between 100 and 300 lines of code at a time and spend 30-60 minutes to review it" [11]. The study also mentions the importance of filtering out both trivial code and thousand-line blocks of code from the reviewing process. Part of their evaluation method looked at the defect density for code, but they did not have any conclusive expectations for what that number should be.

Defect rate is a widely discussed topic. We can define a defect as a variation of the specification which may lead to failure in execution. There are several existing models for predicting defects. A study by Fenton suggests that the expected number of defects increases with the number of code segments but there are also opposing metrics that suggest that larger modules have lower defect densities [13]. Marek Vokac did a study on the the relationship between defects and design patterns [17]. The conclusion of the study is that Observer and Singleton patterns lead to a lot of complexity and higher defect rates while the Factory and Template Method patterns resulted in lower complexity and required less attention [16]. There is little consensus on what the defect rate is for a given program or even how to predict it.

Ideally if we could figure out areas of code that are likely to have defects, those are the areas that Caesar would prioritize for review. However, since the studies seem divided on how to find defect rates, it is unclear how feasible this will be.

3.3 Variations in Code Review

There is existing work done in examining the characteristics of people and how effecitive they are at software engineering tasks. Work by Devito Da Cunha and Greathead [12] shows that personality type is correlated with how effective students are at a code review task. They conduct their study by taking a sample of undergraduate software engineering students, giving them personality tests, and then asking them to conduct a code review task. They show that, in their sample, certain personality

types are more effective at finding bugs in code than others.

Such a problem cannot readily be studied in commercial settings, since it is often not the case that a large number of people look at the same piece of code. However, the setting that Caesar presents is perfect for analyzing this, and similar questions. Because potentially dozens of reviewers are looking at the same piece of code, and we can look at all of the code reviews given by a particular student through the semester, it seems like this data set would be ideal for analyzing variations in code review ability. Caesar, then, would be very relevant for studying this, and similar problems.

Chapter 4

Selecting Chunk Size

One of the variables that changed over the Fall semester was chunk size, where chunk refers to the smallest amount of code reviewed at a particular instance. The amount of code presented to a reviewer may impact how much feedback they leave. Too little code may obscure the point of the code and too much code may overwhelm the reviewer. Since we want to maximize the amount of useful feedback given to students, finding if there was a chunk sized that worked best is important.

In the original design, the system presented users with a single method, and asked them to perform at least one action. We define an action as leaving a comment or voting on an existing comment. Caesar had control over which parts of the code will get attention and feedback. The system was configured to pick out a set of chunks that contained some common elements but differed in their implementation; the hope was that these chunks would contain the important design decisions. As long as the system is intelligent about picking out which areas to review, then it is picking out the substantive areas for review. An alternative is to give reviewers a complete class to review while maintaining the same requirement from users: to perform at least one action. Our results showed that more useful comments were generated when users were presented with full classes rather than methods. Our metric for evaluating success was looking at the types of comments left by reviewers and how the total number of comments differed between the two styles of chunks.

4.1 Motivation

During the Fall 2011 and Spring 2012 iterations of 6.005, the system had two main techniques for picking out chunks. The first method involved partitioning student code into methods, which meant that the size of our chunk was a single method. An indirect consequence of this is that class level declarations and field variables would never be reviewed. With this system, each reviewer would be assigned similar methods from multiple students to review. On average, the size of the chunk was 22 lines.

The other technique was to partition student code into classes, and thus each chunk would correspond to a Java class. This has the advantage that class level declarations and field variables would be visible to reviewers. The average size of the chunk grew to 110 lines but that included more blank lines, getters and setters and import statements. The number of substantive lines was closer to 70. In order to keep the work load similar, we scaled the number of assigned tasks to 10 for method sized chunks and 5 for class sized chunks.

When presented a method, the reviewer may have little context as to how the method fits into the rest of the code. The Caesar interface does provide a "see all code" button which shows all the code the user submitted for that problem set, but it would also navigate the reviewer away from the reviewing page. See Figure 2-2. Ideally, the user should have enough information provided on the reviewing page to give useful feedback to the student. We want to be able to direct the reviewer to the section of code they should focus on and not rely or force the reviewer to read through the entire student's project.

This problem of lacking context comes up more often in problem sets where the students are given wide design latitude, as each student may come up with dramatically different ways to partition work between their methods. An example of such an open-ended problem set is a problem set, `calculator`, given in the Spring semester. In `calculator`, the object of the problem set was to create a calculator that accepted the expression such as 6in/12pt[8]. Students were asked to write a Lexer that

tokenizes the expression and a Parser that interprets and evaluate the expression. Beyond those requirements, students had a lot of freedom in design and implementation. Some of the main design decisions included how to handle invalid expressions, how to recurse into subexpressions, and how to evaluate units. Listing 4.1 is an example of typical code the reviewer is asked to comment on.

Listing 4.1: Parser.findCentralOperatorIndex

```
1  /**Method which returns an index of the central operator.
2   * It finds it by skipping over all parentheses objects and
3   * finds the first operator object.
4   * If there isn't a central operator (it is of the form (a+b)
5   * or (a+b)pt) then it returns -1.
6   *
7   * @param ll is the linked list representation of the
8   *         tokenized expression; must be not empty list.
9   * @return an integer which is the index of the central operator
10  */
11 public int findCentralOperatorIndex( LinkedList<Token> ll ) {
12     for( int i = 0; i < ll.size(); i++ ) {
13         if( ll.get(i).text == "(") {
14             i = findCloseParensIndex( i, ll );
15         }
16 
17         if( ll.get(i).text == "+" || ll.get(i).text == "-" ||
18             ll.get(i).text == "*" || ll.get(i).text == "/" ) {
19             return i;
20         }
21     }
22 
23     return -1;
24 }
```

In 4.1, the method's Javadoc gives a basic overview what the method is trying to do, but there are few comments in the rest of the method. In either the Lexer or the Parser, it is important to check if the parentheses match, because by the time

execution gets to `findCentralOperatorIndex`, an unspoken precondition is that the expression has matching parentheses. In order to get a better sense for what this method is doing, it would be necessary to see the details of `findCloseParensIndex` and also to see how `findCentralOperatorIndex` fits within the rest of the Parser. Seeing only this method, the reviewer would not have enough information to know if the student is making justified assumptions in this code. However, despite possibly missing these design decisions, the reviewer would still be able to comment on stylistic choices made in this method. For example, a regular expression matching would be a better solution for finding the operator and picking a variable name "ll" obscures the purpose of the linked list. The reviewer may be content making these comments without diving in deeper into the student's code. The takeaway is that the reviewer is limited to making local comments and may miss the interaction between two "correct" chunks.

Presenting the reviewer with more code can allow them to get more insight into the problem set, but it can also split the reviewer's attention between many components of the problem set, and in some cases, overwhelm the reviewer. One of the criteria for increasing the chunk size was actually asking reviewers if "[they] felt they had had enough information to do a good job [reviewing]"[1]. Looking at the Feedback Survey responses during this problem set one reviewer wrote:

> "I felt like I had a good amount of information, but sometimes people write helper methods whose specifications I don't know. It would be nice if we could reveal those specifications during review."

Another reviewer wrote:

> "When reviewing MultiUnitCalculator.evaluate, I found myself digging into the student's Parser code to see what the Parser was supposed to return. Generally, I found that the initially-presented code lacked enough context, so I went looking through the complete code for more.".

This feedback showed some discontent with the current chunk size. This information seems to suggest that there some styles of problem sets that could benefit from larger

chunk sizes.

Apart from formal feedback, there were several comments on Caesar where the user complained of not to have enough information to leave useful feedback. Consider for example,

> "I don't know the exact logic of your program but I would imagine that you catch and handle all the exceptions and show an error or something in MultiUnitCalculator so throws shouldn't be necessary."

There is certainly room for improvement. In order to see if longer chunks were what the reviewers wanted, halfway through Fall 2011, the chunk size was changed from a single method to a single class.

4.2 Results

In order to see the effect of changing the chunk size, we can look at the type of comments generated for each of the problem sets. We will exclude both iterations of the problem set `intro` from this analysis, because the problem set was much shorter, and contained no interaction between different methods. For the other problem sets, a random set of 125 comments were annotated, and were tagged as either constructive, unconstructive, or other.

We can try to eliminate some of the subjective bias by narrowing down the characteristics of each of the types of comments. We will consider a *constructive* comment as one that:

1. asks the submitter to add documentation
2. change a variable name
3. suggest an alternative implementation or design
4. points out that submitter's code will not work in a specific case

An *unconstructive* comment is one that says the code looks good. Anything that does not fit into this criteria will be tagged as *other*. When tagging comments, in an

attempt to eliminate bias, we will not reveal which problem set the comment came from. Although this analysis is still subjective, it is likely to give us more insight than if we tried to programmatically tag comments.

4.2.1 Examples

Some examples of constructive comments are:

> "Could name it better, also probably wasn't necessary to make it an instance variable."

> "this could have been put inside "for (int j = digits.length...)" to make things more efficient."

Some examples of unconstructive comments are:

> "Creating this state enum was a good idea!"

> "nice concise code"

> "you can have this constructor call this(10) to reduce duplicate code"

Some examples of "other" comments include:

> "Is there something wrong with tab characters?"

> "No code to review."

4.2.2 Data

Figure 4-1 shows the relative number of each type of comments that appear in each problem set in Fall and Spring. The number of comments examined was 125 per problem set. Based on the percentage of constructive comments, we can extrapolate the total number of constructive comments for each problem set. In Table 4.1, we show the average number of constructive comments per student. To further visualize the average number of comments, see Figure 4-2.

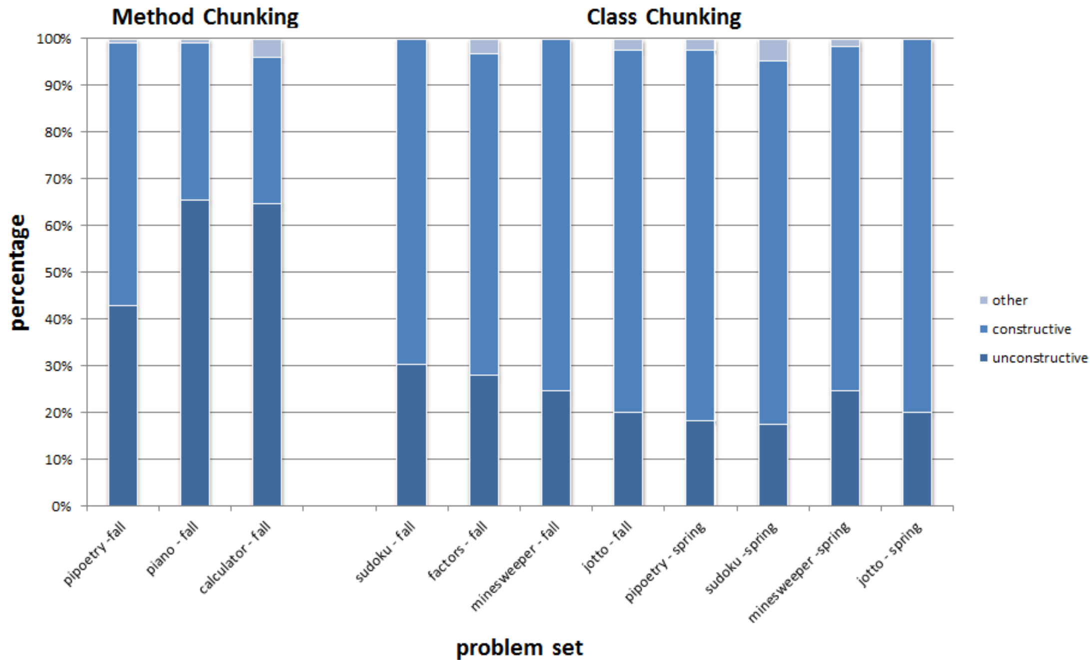

Figure 4-1: Stacked bar plot of types of comment.

Semester	Problem Set	Chunk Type	Comment Type: Unconstructive	Comment Type: Constructive	Comment Type: Other	Average Constructive Per Student
Fall	`pipoetry`	method	56	68	1	5.2
	`piano`	method	82	42	1	2.8
	`calculator`	method	81	39	5	2.6
	`sudoku`	class	38	87	0	7.0
	`factors`	class	35	86	4	4.6
	`minesweeper`	class	31	94	0	4.7
	`jotto`	class	25	97	3	4.3
Spring	`pipoetry`	class	23	99	3	6.8
	`sudoku`	class	22	97	6	9.0
	`minesweeper`	class	31	92	2	9.0
	`jotto`	class	25	100	0	8.2

Table 4.1: Statistics on comment type for each problem set.

The percentage of unconstructive comments is higher when chunk size is set to methods. Reviewers are asked to make comments on a variety of methods, some of which are only a few lines, and do not contain any complicated components. Class-sized chunks show a dramatic increase of constructive comments.

However, this diagram does not tell the complete story. When chunks were meth-

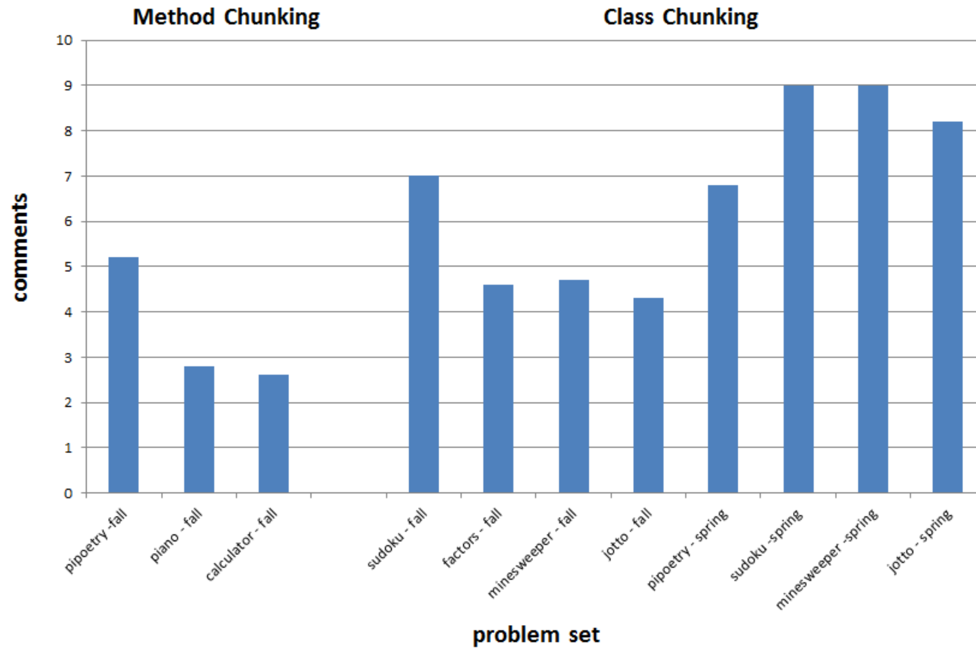

Figure 4-2: Bar plot of average number of comments per student.

ods, the number of assigned tasks was set to 10 but this number dropped to 5 when chunks were changed to classes. Since students were asked make at least one comment on a chunk, being assigned more chunks could result in more comments being produced overall. In order to objectively see if the total number of useful comments in the system increased, we can look at the total number of constructive comments.

In Figure 4-2, we can see that Fall problem set `pipoetry` does not fit the rest of the trend. However, if we consider the nature of the problem set, this may not be surprising. Problem set `pipoetry` asked students to find English words in the digits of Pi. The procedure for finding words was separated into 4 methods that were each in their own class[7]. The specification for each method was provided by the staff, therefore any design decisions were isolated to the method itself. If the chunk size was set to class size, the user would have largely seen the same amount of code with some exception of students that used helper methods. The rest of the data suggests users leave more useful comments when they are presented with larger chunk sizes. One explanation for this data is that when the user is presented with more code, he

or she can choose which parts of the code require attention and leave comments on those parts. The methods that look fine simply do not receive any feedback.

In Figure 4-2 we can see a positive trend between class-sized chunks and number of constructive comments. Our data, combined with reviewer feedback that this change was desired, justifies permanently switching to class sized chunks.

4.2.3 Potential Shortcomings

Our analysis does not take into account that students may have learned better code reviewing techniques and were naturally more inclined to stay away from unconstructive comments later in the semester. As students become more familiar with Java, they are likely to offer up their own perspective on how to solve the problem. This could serve to explain why halfway through the semester the number of unconstructive comments dramatically dropped. The other flaw in this analysis is that, in the later problem sets, students had more room to make design decisions, which is something reviewers could discuss in their code review.

Having a control group would have helped eliminate some of these biases. Ideally the experiment should have been set up with students being assigned class and method sized chunks during the same problem sets. This would allow us to compare how comment substance varied at that particular student's coding experience level.

Chapter 5

Chunk Filtering

Caesar implements a task routing mechanism to automatically and dynamically allocate chunks to reviewers. When assigning chunks, the system considers reviewer and chunk characteristics. Chunk characteristics have varied throughout the life of the system. Examples of chunk characteristics that the system cares about include number of student written lines, maximum branching depth, and whether the chunk only contains unit tests.

One of the challenges of the system is that the number of reviewers is not known ahead of time. Although we can roughly predict the number of staff and students that will be participating in code review, the number of alums participating may fluctuate significantly. For example, the number of alums has varied from 0 to 30. As a result, we use a greedy algorithm to assign chunks by always picking out the most optimal chunks to review. Chunks are assigned on a first come first serve basis.

Another constraint on the system is that any chunk characteristics that routing uses to order chunks must be computed ahead of time. Since the system only assigns tasks after the reviewer logs into the system, the reviewer will not want to be waiting for more than a second to receive their tasks. The other limitation is that after the students turn in a problem set, reviewing must open within two hours of that time; ideally even less. Assignments are given with strict deadlines and students need to get timely feedback before their next problem set.

5.1 Partitioning

In order to get code into the web application, we use a preprocessor to parse students' submissions. After the problem set is handed in, the preprocessor crawls the latest copy of the problem set from the Subversion repository, and partitions each student's submission into chunks. The actual partitioning task is handled by the Eclipse Java Development Tools Core Component. The Eclipse tools component parses student codes into of classes, methods, for loops and so on, thereby creating abstract syntax trees which we then traversed with a visitor that creates chunk objects. The visitor learns different characteristics about the chunks, including whether they're test files, max number of branching factors, and other relevant information. Following the partition, the system runs Checkstyle[2] over all the chunks to automatically generate comments whenever Java style rules are violated. Checkstyle comments then get uploaded to the web application. Figure 2-7 shows the path from the assignment to the reviewing process.

5.2 Preprocessing

The implementation varied of the preprocessor from partitioning student code into methods and constructors to classes. When chunk size was set to methods, the system would remove chunks that were string-identical throughout 80% of the students' submissions. This ensured that staff-provided code would not be reviewed. In this design, the system did not need to be explicitly given the starter code; it could simply infer it. The problem with this design is that if a student made even a minor modification to staff code, that entire chunk would get reviewed, and the reviewer would not have any visual indicator that the majority of that code was staff written. In practice, there were also times when staff would issue an errata to the problem set, and half the students would have new version of the code while others would have the old code; both versions would get reviewed even though neither version was original.

Changing the chunk sizes to classes required a different approach to picking out

staff code. There were many cases where some parts of the classes were defined by staff but some methods were asked to be implemented by the students. Looking at redundant chunks would not help in those cases. Due to the new chunking method, the system has to be supplementally fed the starter code for each problem set. The system now also crawls over starter code and partitions it into classes. Whenever the class name exists in the starter code and the student implementation, the preprocessor checks to see if any lines in the student implementation also appeared in starter code. This information is then uploaded to the web application in order for the reviewing interface to display which lines of code were modified by the student.

5.3 Changes to Display

When chunk size is a class and some lines are staff provided, we want to make sure student lines of code stand out. As a result, we chose to depict student written lines on a gray background and staff written lines on a white background. See Figure 5-1 for a demonstration of this in action.

The system does not prevent the reviewer from leaving comments on the staff provided sections, but it does try to highlight sections of code that the student wrote. Other code review systems highlight versioning changes in the same manner, for example this method is consistent with the native code review system on Github[4].

5.4 Routing Details

The original task routing algorithm tried to assign similar chunks of code to reviewers. The preprocessor would cluster similar code and assign each reviewer chunks to review from the same cluster. It did so by implemented a version of the robust winnowing algorithm for document fingerprinting developed by Schleimer et al. [14] for the purposes of measuring code similarity. A fingerprint for a chunk is an MD5 hash of an n-gram that occurs in the chunk contents[15]. The robust winnowing algorithm searches the set of potential fingerprints for a chunk and selects a small subset of

```
 6  public class WordFinder {
 7      /**
 8       * Given a String (the haystack) and an array of Strings (the needles),
 9       * return a Map<String, Integer>, where keys in the map correspond to
10       * elements of needles that were found as substrings of haystack, and the
11       * value for each key is the lowest index of haystack at which that needle
12       * was found. A needle that was not found in the haystack should not be
13       * returned in the output map.
14       *
15       * @param haystack The string to search into.
16       * @param needles The array of strings to search for. This array is not
17       *                mutated.
18       * @return The list of needles that were found in the haystack.
19       */
20      public static Map<String, Integer> getSubstrings(String haystack,
21                                                       String[] needles) {
22          Map<String, Integer> output = new HashMap<String, Integer>();
23
24          for (int i = 0; i < needles.length; i++)
25          {
26              String n = needles[i];
27              if (haystack.contains(n))
28              {
29                  int index = haystack.indexOf(n);
30                  output.put(n, index);
31              }
32          }
33
34          return output;
35      }
36  }
```

Figure 5-1: Code section containing student and staff lines.

those fingerprints in a consistent way. For our implementation, we use n-grams of size 10 and a window of size 20 for winnowing. In order to cluster the chunks we measure the distance between every two chunks. We define distance as the number of unique fingerprints to each chunk.

In practice, this typically resulted in a reviewer being assigned multiple student implementations for the same method specification. When the chunks were methods, and the average size of each chunk was about 22 lines, the clustering algorithm ran in about an hour for 3500 chunks amongst 180 students. When the chunks changed to be class-sized and 140 lines, the algorithm took over four hours to complete. The bottleneck was in taking n-grams of size 10 for every chunk. The clustering algorithm also performed poorly in terms of output when given large amounts of staff code or when fingerprints were being generated for irrelevant parts of the student code. In those cases the clusters did not help reviewers. For these reasons we chose to do away with clustering.

Clustering code and assigning chunks from clusters was what chunk routing relied

on. Since we no longer have clusters, Caesar needs to prioritize chunks based on some other chunk characteristics. One of the ideas was simply to prioritize chunks that contained the most lines of student code. The logic behind this heuristic was that more lines would mean more room for error. The concern with this approach was that any classes that are completely defined by the student would get the most attention. This was the task routing algorithm for problem sets 4-7 (`sudoku`, `factors`, `minesweeper`, and `jotto`) in Fall 2011.

In order to evaluate our task routing algorithm for problem sets 4-7, we will look at the total number of student comments and students words entered in the system, see Table 5.1. In Chapter 4 we looked at the percentage of constructive comments entered for different size chunks. However, for problem sets 4-7, the percentage of constructive comments was relatively consistent, so we look at the comments as a whole.

Problem Set	Average comments per student	Average words per student
`sudoku`	10.6	156
`factors`	6.95	93
`minesweeper`	6.70	90
`jotto`	5.95	75

Table 5.1: Results of varying chunk sizes and new routing algorithm.

Figure 5-2 shows the results of the routing. The number of words entered follows the same pattern as the number of comments and both statistics suggest that routing performed much better for Problem Set 4. There are several theories that could explain this phenomenon.

1. Problem Set 4 was highly templated and the classes with the most student lines happened to be the most relevant.

2. Students devoted less time to reviewing as the semester progressed. See Figure 5-4.

3. Students made fewer mistakes in later problem sets leading to less for the reviewers to do. See Figure 5-3.

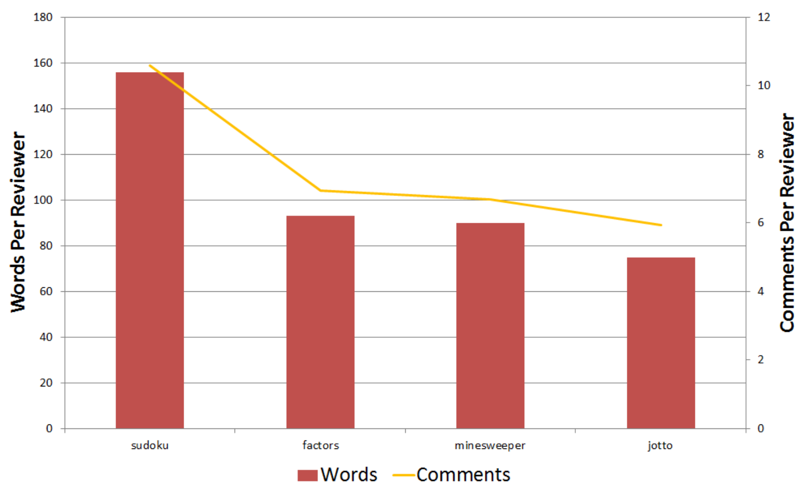

Figure 5-2: Bar plot showing average number of words and average number of comments left by students.

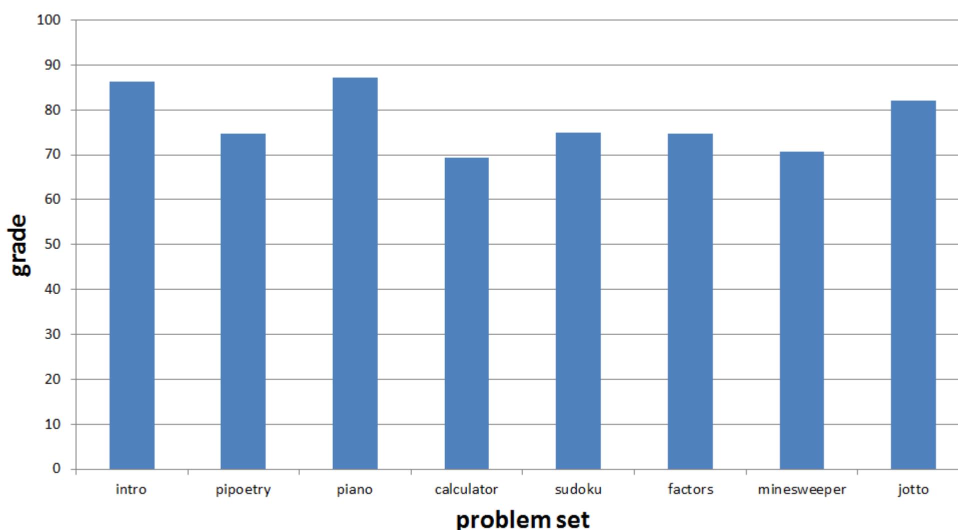

Figure 5-3: Bar plot of average problem set grades.

Figure 5-4 shows the total review time for problem sets. Review time is calculated by finding the average time between a student opening a task and performing their last action on that task. Outliers are removed from calculation. The average time per task is then multiplied by the average number of tasks performed by a student. From Figure 5-4, we can see that the total review time for a problem set did somewhat dip for the last three problem set. However, because this is only a correlation, we do not

Semester	Problem set	Average comments per student	Average words per student
Fall	`sudoku`	10.6	156
	`minesweeper`	6.70	90
	`jotto`	5.95	75
Spring	`sudoku`	12.2	185
	`minesweeper`	12.4	210
	`jotto`	10.3	116

Table 5.2: Results of varying chunk sizes and new routing algorithm.

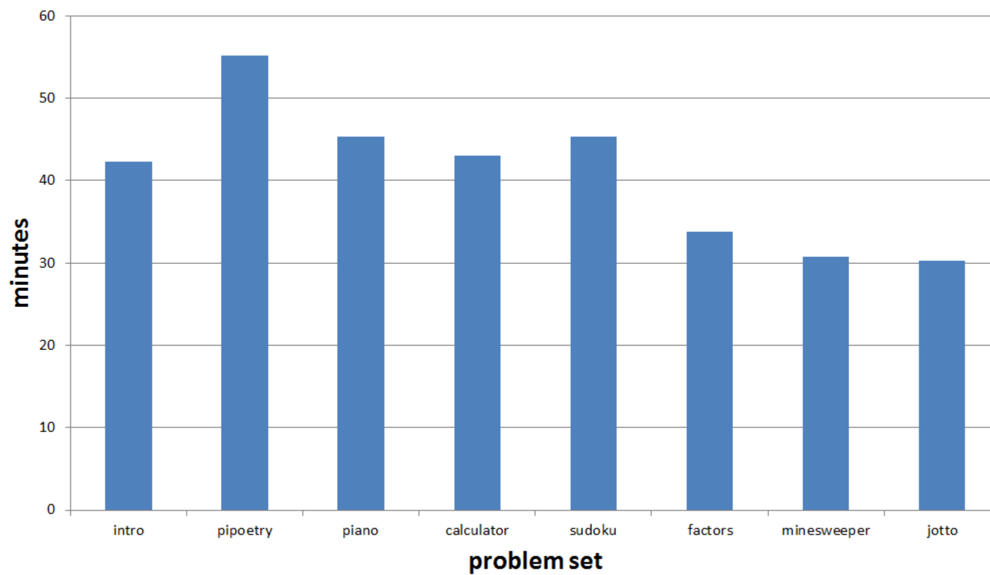

Figure 5-4: Bar plot of review time.

know why students were spending less time reviewing. It could be because the chunks they were given did not require much reviewing, or it could be they were simply less willing to devote their time to code review.

Figure 5-3 shows the average grade in each problem set. If we consider that making mistakes in the problem set allows reviewers to make comments, it would suggest that there was still plenty to comment on in later problem sets. Regardless of what the real reason was, we can still try to improve our chunk selection algorithm to try to increase student participation in review.

5.5 Routing Based on Branching Depth

Between Fall and Spring, we experimented with changing the routing settings. In Fall we prioritized chunks strictly on student lines. In Spring we only took chunks that had the minimum number 30 student written lines and prioritizing chunks to review based on max branching depth, where branching depth is the maximum number of loops and conditionals used in a program. The study by Fenton showed that, in some cases, more complex code, where complex is defined roughly as branching depth, leads to a higher defect rate. This was refuted as an effective metric in all project types[13], but it had the potential to be effective in our case. We ran this experiment on problem sets `sudoku`, `factors`, `minesweeper`, and `jotto`. In both semesters, once chunks were class-sized, test files were not reviewed.

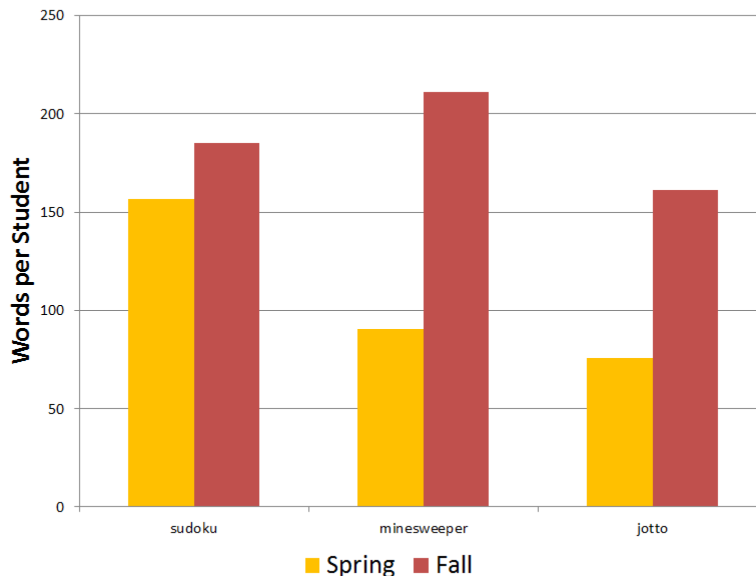

Figure 5-5: Fall versus spring average number of words per student.

On average, in Spring we see the number of comments increased by about 20% and number of words by 40%. Figures 5-5 and 5-6 show a side-by-side view of words and comments for each of the problem sets that were in Fall 2011 and Spring 2012.

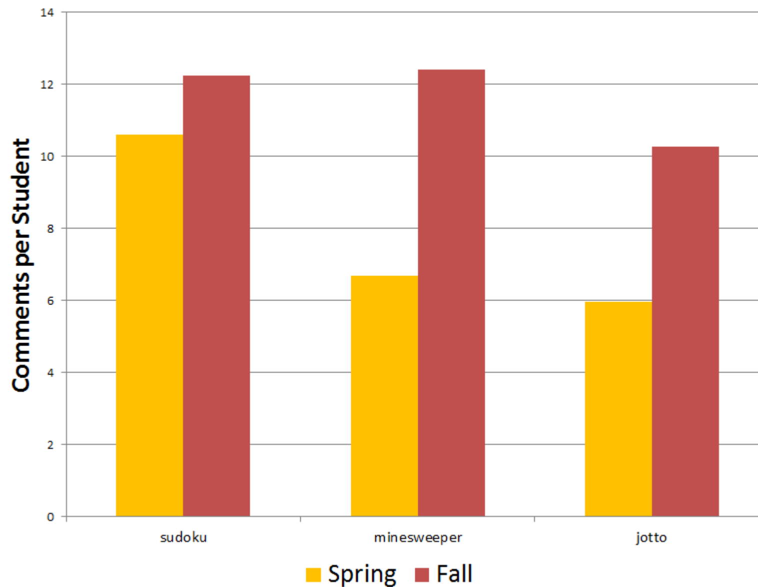

Figure 5-6: Fall versus spring average number of comments per student.

5.6 Shortcomings

Routing can not take all the credit for this increase. In Spring 2012 students had only half as many problem sets and each problem set had a beta and a final submission. The beta submission was the one code reviewed and students were asked to implement reviewer suggested changes in their final submission. Students had to heed the advice from code review comments and this could have incentivized reviewers to do a better job.

In addition, after receiving feedback that some reviewers wanted to perform more code reviewing then what was assigned to them, we built in a "Get More Tasks" feature. This button is only visible to reviewers that do not have any outstanding tasks and it is only available while the code reviewing is going on for the problem set. Due to our on demand task assigning algorithm, this change did not require any significant design changes. Only about 2% of the reviewer population use this feature. A small increase in comments and words may have come from this bonus feature but it's negligible.

Chapter 6

Routing Interface

Originally, all the settings for task routing were hard coded into the routing algorithm, and only a Caesar developer would be able to make changes to them. To make the system more versatile, we want to easily tune the routing settings for different problem sets. To accomplish this we built a task routing interface. The task routing interface allows a staff member more control over how many reviewers should be assigned to a chunk, what classes get reviewed, how they are prioritized, and the minimum number of lines for a chunk to be considered for review.

Starting with Fall problem set `pipoetry`, not all of the student code could be reviewed. Some 6.005 staff members showed discontent that much of the routing was a black box to them, and they were not satisfied releasing so much control over the problem set to the system. A problem set may have special circumstances or the staff may decide that tests should be reviewed. The reviewing interface needed to explain reviewing as well as allow the staff to easily select which chunks to review.

6.1 Components of the Interface

The first hidden component of review is figuring out the reviewing capacity. The number of chunks that will be reviewed is dependent on how many students and alums are participating in code review, and how many reviewers should be assigned to each chunk. Our reviewing interface, as seen in Figure 6-1, makes it easy to predict

how many chunks can be reviewed. In order to help the user predict these numbers, the system uses past problem sets to give baseline values. These numbers are in purple in Figure 6-1. Changing any number in the text fields will automatically update the relevant values, and in particular, update the number of classes to be reviewed.

Settings for ps3-beta.

Assign 2 non staff users (students and alums) to each class for review. Staff will serve as quality control and be assigned as a 3rd reviewer to as many classes as possible.

For the reviewing process we can expect:

Baseline values for ps3-beta in *purple*.

199 *199*	students	reviewing	5 *5*	tasks each.	199 x 5 = 995 tasks total.
19 *1*	alums	reviewing	3 *3*	tasks each.	19 x 3 = 57 tasks total.
15 *15*	staff	reviewing	10 *10*	tasks each.	
					/ (2 tasks/class) = 526 classes

At most 526 classes will reviewed.

Figure 6-1: Reviewing interface showing how many chunks can be reviewed.

The next important setting to consider is the minimum student lines to have in a chunk for it to get reviewed. Having one or two student lines is likely not enough for a reviewer to comment on, but in some cases perhaps 10 or 20 student lines is enough. This number may be dependent on the problem set, and needs to be controlled by the staff. In order to help the staff judge that minimum, our routing interface shows a graph of student lines. The x-axis shows the number of lines and the y-axis shows the number of student chunks that wrote that many lines. Figure 6-2 shows a typical graph that the staff may see. Making changes to the text field will move the vertical line seen on the graph.

The other important setting has to do with which chunks should be reviewed and how they are prioritized. In highly templated problem sets, the staff may know which classes contain important design decisions and need to be reviewed. In other cases, student defined classes may be more important to review. Since it would be

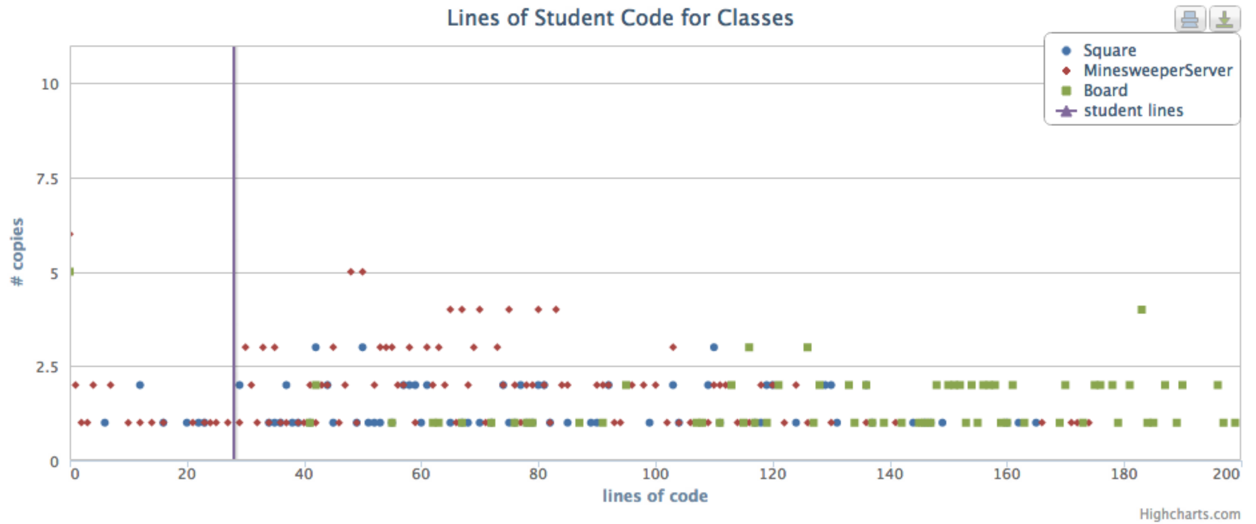

Figure 6-2: Reviewing interface showing distribution of student lines.

overwhelming to list all the names of student defined classes, they are simply grouped together under the term of "StudentDefinedClasses". The user can check or uncheck if those classes should be reviewed and the staff simply drags them order the classes by priority. The number next to the chunk names is how many instances of those chunks will be reviewed based on minimum number of student lines defined in the interface. The number of each chunk dynamically changes with relation to that input. The bottom also displays the number of chunks scheduled for review. Although the system will not prevent the staff from scheduling significantly more or significantly fewer tasks than the reviewing capacity, the numbers do provide more information for the staff.

6.2 Responses to Interface

This interface received positive feedback from a 6.005 Spring 2012 TA during a user test. His main priorities were being able to select which chunks would be picked for review, and being able to set the priority order for the chunks. He said the interface

Check which classes should be reviewed
Drag to prioritize reviewing order.

- ☑ Square, 88
- ☑ MinesweeperServer, 168
- ☑ Board, 106
- ☑ StudentDefinedClasses, 194
- ☐ BoardTest, 115
- ☐ StudentDefinedTests, 80

The following classes are too short
(in terms of student lines):

556 classes scheduled for review.

Figure 6-3: Reviewing interface for prioritizing chunks.

easily allowed him to do that. This interface allows the system to function without a developer, and does not require constant digging into the code to make configuration changes.

Chapter 7

Conclusion

7.1 Future Work

The foundation of Caesar is already providing a robust platform for students to receive feedback on their work, but there is a huge space for improvements on top of the basic platform.

Hashtags similar to those found on Twitter and other social platforms are currently being implemented for user comments. This allows students in the class to dynamically tag comments, so comments that are similar to each other can be grouped together and easily browsed. For example, a student can tag a comment with a tag like `#deadlock`, which could suggest to the code author that there is a potential deadlock situation in a segment of code. The author can click on that hashtag to see all of the other comments tagged as deadlock for the current assignment. This lets the student jump to other student code which has the same problem, which could be a useful resource for debugging.

Part of the motivation of Caesar is that, because 6.005 has such high attendance at MIT, it is infeasible for the limited course staff to look at every student submission. For the same reason, all of the problem sets were mostly graded with automated unit tests, whenever possible. For each problem set, the course staff writes a testing suite that tests student code, and a large component of the student grade is based on whether or not each unit test passes or fails. Currently, this component of the class

operates independently of Caesar, but lots of value can be gained if the two systems were integrated. Unit test dashboards are commonplace in industry [5], and being able to see the pass or fail status of unit tests alongside code review would provide benefits for students. Given that a test fails, it could be a signal for reviewers as to where to find faults in the code. This would encourage more constructive feedback for students.

Because Caesar offers students the ability to reply to existing comments, there are often interesting discussion threads that could provide other students insights into the nuances of a particular piece of code. The ability to highlight such threads to other students could be integrated into the system. In addition to showing the student what code they have left to review, the system can also point them to lively discussions on the dashboard page. This would give students access to real code, along with relevant commentary, which could serve as valuable examples for how to write constructive comments.

In industry, it is common for people to be able to mark each comment on their code as resolved or unresolved. This lets reviewers easily scan what parts of the code are still being improved, versus issues that have already been resolved. This functionality is not built into Caesar yet. The newest system of beta versus final submissions would benefit greatly from such a system; presently, course staff has to manually scan the student code to verify that all the comments have been addressed. If students could manually assert themselves that a comment has been addressed, and a diff could be generated by Caesar to show that code has changed, this manual process can be automated.

It is common for students to make identical mistakes in a problem set. For example, students may use paradigms that are no longer in the Java mainstream, like Iterators. The same comment could be applied to multiple students with the same problem and there should be an easy way within the system for reviewers to do that.

7.2 Summary

This work presents Caesar, a system for distributed code review in a classroom context. Existing code review systems are designed for commercial use, and are not generally applicable for the classroom setting. Caesar is designed to process code that is a hybrid of student contributions and staff-provided boilerplate, and presents that code to reviewers in a usable web interface. The system is designed to optimize the amount of useful feedback that is generated for students in the class. The primary contributions of this work are the algorithms for partitioning student code, and routing tasks to reviewers.

Partitioning student code is a challenging task, as there is a delicate tradeoff to be made in figuring out what, and how much code to send to each reviewer. Too little code sent to each reviewer has the risk of not providing enough context to generate helpful comments. Too much code sent to each reviewer risks reviewers being overwhelmed and unfocused. This work presents experimental results that examines the amount of productive comments left with each partitioning method, and sheds light on the amount of context necessary to properly evaluate Java code. The conclusion that is supported by the evidence here is that presenting class-sized chunks to reviewers is much more effective at soliciting useful feedback, as opposed to method-sized chunks.

Routing tasks to reviewers is the other key contribution made by this work. This is a hard problem, because each problem set necessarily gives different amounts of flexibility to students completing the assignment, and may require different heuristics on how to properly allocate work to reviewers. In addition, the set of reviewers changes between each problem set, and the system has to predict how much to assign each reviewer in order to give each student adequate feedback. Experimental results show that routing based on branching factors as well as having a minimum number of lines for reviewing a chunk performs better than strictly using the number of student lines.

The experimental results given by this work shows that by using Caesar, students

in a undergraduate-level software engineering class are able to give and receive constructive feedback on the code they submit for assignments. By effectively chunking and routing student code, we can increase the amount of useful feedback that is left for students. This is an encouraging result for preparing students for real-world software development environments.

Bibliography

[1] Caesar feedback survey. https://spreadsheets.google.com/spreadsheet/viewform?formkey=dGNyMlRVTGNTbXFZazA5bDRPd25sZFE6MQ.

[2] Checkstyle. http://checkstyle.sourceforge.net/.

[3] Gerrit code review. http://code.google.com/p/gerrit/.

[4] Github code review. https://github.com/features/projects/codereview.

[5] Lithium's unit testing framework. http://lithify.me/docs/manual/quality-code/testing.wiki.

[6] An open source app: Rietveld code review tool. https://developers.google.com/appengine/articles/rietveld.

[7] Problem set 1: Pi poetry. http://web.mit.edu/6.005/www/fa11/psets/ps1/assignment.html.

[8] Problem set 3: Calculator parser. http://web.mit.edu/6.005/www/fa11/psets/ps3/ps3.html.

[9] Reviewboard. http://www.reviewboard.org/.

[10] A. Alali, H. Kagdi, and J.I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 182–191. IEEE, 2008.

[11] J.A. Cohen. *Best Kept Secrets of Peer Code Review*. Printing Systems, 2006.

[12] A.D. Da Cunha and D. Greathead. Does personality matter?: an analysis of code-review ability. *Communications of the ACM*, 50(5):109–112, 2007.

[13] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[14] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[15] M. Tang. *Caesar: A Social Code Review Tool for Programming Education*. PhD thesis, Massachusetts Institute of Technology, 2011.

[16] D.A. Trytten. A design for team peer code review. In *ACM SIGCSE Bulletin*, volume 37, pages 455–459. ACM, 2005.

[17] M. Vokác. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5(1):139–157, 2006.