Search Tools for Scaling Expert Code Review to the Global Classroom

by

Abigail Klein

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

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author . . . . . .
Abigail Klein
Department of Electrical Engineering and Computer Science
August 6, 2015

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

Accepted by . . . . . .
Professor Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Search Tools for Scaling Expert Code Review to the Global Classroom

by

Abigail Klein

Submitted to the Department of Electrical Engineering and Computer Science on August 7, 2015 in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science.

Abstract

This thesis aims to answer the question "How can teachers of online classrooms give more qualitative feedback to students?" We narrow the scope of this question to an online software engineering class in which a major component is code review. We built two search tools that give teachers better coverage of student code. The first tool, Comment Search, allows students and staff to reuse any comment they previously wrote when reviewing another student's code. Staff can reuse any comment written by any staff member as well. After deploying Comment Search in a classroom for a full semester, we found that students and staff used this tool to write higher quality comments. We also found that many reused comments were about similar patterns in code. This inspired the second tool, Code Search, which allows teachers to search for sections of student code that contain a desired pattern. Preliminary results of Code Search are promising: for the queries that Code Search is built for, Code Search returns nearly all relevant results. Together, Comment Search and Code Search offer teachers the ability to give meaningful comments to many more students than otherwise possible.

Thesis Supervisor: Robert C. Miller
Title: Professor

Acknowledgements

First and foremost, I want to thank my advisor and mentor, Rob Miller. Rob put in an enormous amount of time to help me when I needed it, coming in on the weekends to meet with me and sitting one-on-one with me weekly if I wanted. He was accommodating of my unorthodox schedule due to track practices and competitions. What's more, he was patient and understanding when I most needed a mentor outside of the workplace. Most amazing is how he balances mentoring all of his graduate students with teaching two classes a term, being the Educational Officer, and being the EC Housemaster. I am very grateful to have been able to to learn from him.

Next, I want to thank Max, Elena, and Rob for their help with designing the Code Search pattern language. Despite it being difficult to coordinate a meeting between 4 very busy people weekly, it was really fun. And Max, sorry about the triple-character operators. We'll go with emoji's next time.

Thanks to everyone in the UID group for their help during pair research (every week, we help each other out with research blockers). And thanks for being a great group of people with a great sense of humor!

Last but not least, I want to thank my friends and my family. To my friends, in particular the pole vaulters, who are my family here at MIT, thank you for being some of the most caring, funny, interesting, crazy, and altogether amazing people I had the good fortune to spend my time at MIT with. And to my family, thank you for being unconditionally loving and supporting of me.

Contents

1 Introduction 11
1.1 Motivation 11
1.2 Comment Search 12
1.3 Code Search 13
1.4 Contributions 14
2 Related Work 15
2.1 Existing Tools for Powergrading 15
2.2 Clustering Code 16
2.3 Code Search 16
2.4 Caesar 17
3 Comment Search 18
3.1 Motivation 18
3.2 User Interface 20
3.3 Implementation 24
3.4 Evaluation 25
3.4.1 Usability Evaluation 25
3.4.2 Effectiveness Evaluation 26
3.4.3 Applications for Code Search 29
4 Code Search 31
4.1 Pattern Language 31
4.1.1 Primitives 31
4.1.2 Operators 32
4.1.3 Return Type 35
4.2 Implementation 35
4.2.1 Query Parsing 35
4.2.2 Code Parsing 36
4.2.3 Evaluation 37
4.2.4 Scoring 39
4.2.5 Display 42
4.3 Evaluation 43
4.3.1 Qualitative Evaluation 43
4.3.2 Information Retrieval Accuracy 48
4.3.3 Performance 50
4.3.4 Limitations of Code Search 50
5 Discussion 52
5.1 Comment Search 52
5.2 Code Search 52

5.3 The Comment Search/Code Search System 53
6 Future Work and Conclusion 54
6.1 Future Work 54
6.1.1 Comment Search 54
6.1.2 Code Search 54
6.2 Conclusion 56
7 References 57

List of Figures

Figures

Figure 1. The Comment Search interface with 3 similar comments displayed 12
Figure 2. The Comment Search context bubble 13
Figure 3. Caesar's code reviewing interface 17
Figure 4. Caesar's comment interface 20
Figure 5. A newly-saved comment in Caesar 20
Figure 6. The Comment Search interface with three similar comments displayed 21
Figure 7. Comment Search context bubble 22
Figure 8. Comment Search text area after user selects a comment 22
Figure 9. Comment Search for a reply 23
Figure 10. Data processing pipeline of Fullproof scoring engine 25
Figure 11. Chart of comment reuses using Comment Search 27
Figure 12. Code Search pipeline 35
Figure 13. Context-free grammar for Code Search pattern language 36
Figure 14. Pseudocode for evaluation of Code Search 37
Figure 15. Code Search results 42

Tables

Table 1. Common mistakes found by code reviewers and sample queries
that describe the mistakes 14
Table 2. Similar comments written by Caesar code reviewers 19
Table 3. Type of apparatus used to interact with the system 25
Table 4. Example of similar comments and their correspondingly similar lines of code 29
Table 5. Pattern language operators 32
Table 6. Code Search results for repeated code 44
Table 7. Code Search results for while loops that should be for loops 45
Table 8. Code Search results for misunderstood method specifications 46
Table 9. Code Search results for a bug in angle calculation 47
Table 10. Precision values for 10 sample queries 48
Table 11. Coverage ratios of code review with Code Search compared to without 50

Chapter 1 Introduction

Online classrooms are becoming increasingly popular with the rise of edX, Coursera, and Khan Academy. They have many benefits, namely: (1) they are free and available to the entire world, (2) they can be catered toward individual students more easily, and (3) they can be delivered to tens of thousands of students at once. One major limitation of MOOCs is that students and faculty interact minimally, both because students are learning remotely and because there are too many students for faculty to interact with all of them. This means that, at least currently, most assignments are graded for correctness, rather than for style, clarity, and other aspects of quality that are hard to measure automatically. In an English class, for example, this means that students might answer multiple choice questions rather than writing essays. In a software engineering course, this means that code is graded for correctness, rather than for style. Clearly, without qualitative feedback, MOOCs do not offer the same value of education as a traditional classroom. **The vision of this thesis is to create tools to support staff of massive open online classrooms (MOOCs) and large on-campus classrooms to give qualitative feedback to computer science students.**

1.1 Motivation

6.005 is an introductory course to software engineering for MIT undergraduate computer science majors. One of the unique parts of 6.005 is its code-reviewing component. Students use a platform called Caesar [1]. Students submit assignments to Caesar, and Caesar assigns chunks of code to other students, staff members, and volunteers to review. After the reviewing period ends, students can read comments on their code, revise their programs, and re-submit them.

The value of code review in 6.005 is three-fold. First, students receive comments on their own code. Many 6.005 students lack experience writing large software engineering projects, so receiving critiques on their coding style and technical abilities is invaluable. Second, students are exposed to different coding styles as they review their peers' code, which is critical to a teaching them what both good and bad code looks like. Third, students gain experience participating in code reviews. Code review is very common in industry, and it is critical that students are prepared for this. Even though students gain a lot from reviewing other students' code, they sometimes make mistakes in their comments. For this reason, Caesar also uses expert reviewers: staff members and experienced volunteers to give quality assurance.

Because 6.005 has only 200 students per semester, expert reviewers can read a large fraction of the students' code. In a few years, 6.005 will be offered in a massive open online classroom (MOOC) to approximately 20,000 students. When this happens, 6.005 will need tools that increase expert coverage of student code.

Many comments in Caesar identify similar problems in code. This suggests that students make similar mistakes in their assignments. We propose two tools that will allow staff members to take advantage of this fact and use the same comment on similar programs: Comment Search and Code Search. Importantly, these tools are meant not to replace expert participation but to augment it. Using Comment Search and Code Search, experts will be able to reach more students with the same high-quality feedback that they give when reviewing code with Caesar.

1.2 Comment Search

Comment Search is intended to help teachers and students alike write higher quality comments about code. Rather than needing to write a unique comment for every line of code they wish to critique, students and teachers can reuse previously written comments.

The figures below show the Comment Search interface. As a code reviewer writes a comment, Comment Search suggests the most relevant previously written comments: those that contain the most matching keywords (Figure 1). Comment Search offers contextual information about the older comment, such as the author, author's reputation, and the lines of code to which the previous comment referred (Figure 2).

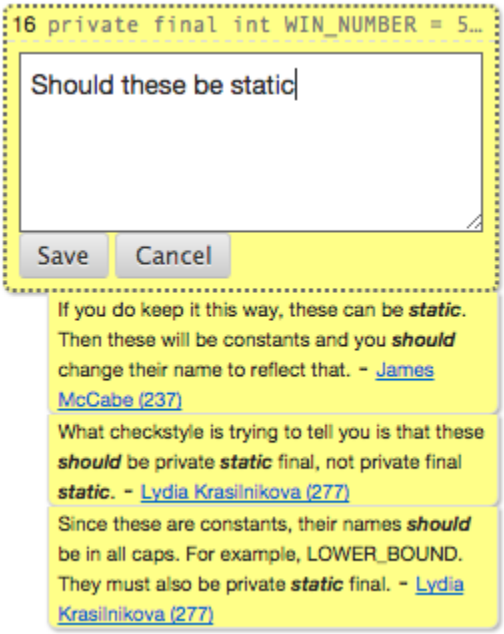

Figure 1. The Comment Search interface with 3 similar comments displayed.

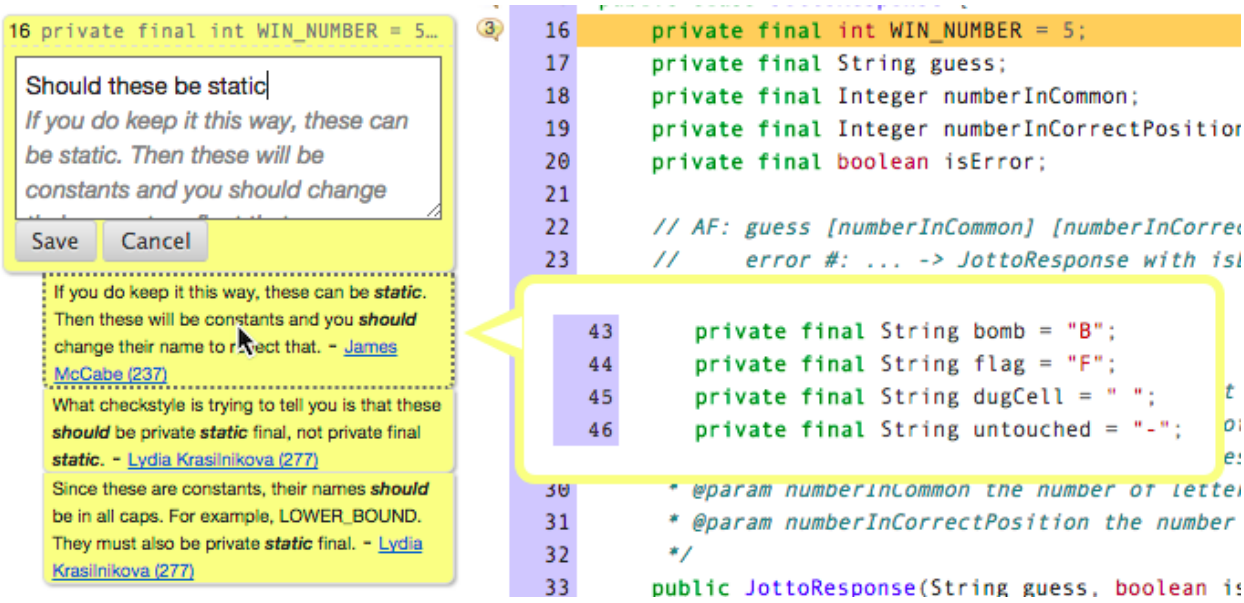

Figure 2. The Comment Search context bubble. When code reviewers hover over a suggested comment, they can see a context bubble containing the lines of code that the original comment referred to. The italicized text that appears in the textbox shows where the suggested comment will appear in the textbox if the user selects it.

I implemented Comment Search for Caesar and we tested it for one semester of 6.005. Over the four months, five problem sets, and 33,483 comments written by staff and students, 1% of comments written that semester were reused using Comment Search. Code chunks are assigned randomly for review and each reviewer only reviews ten chunks a session, so the fact that authors found many repeated mistakes supports the need for this tool.

1.3 Code Search

The results of Comment Search confirmed that many students make similar mistakes in their code. The next step was to create a tool where staff members could comment on *all* of the same mistake at once. Code in software engineering classes is too long and complex to form meaningful clusters, so automated code clustering techniques are not a viable option. Instead, Code Search allows expert Java programmers to search for patterns in code. This keeps the expert Java programmer involved in the process and takes advantage of their intelligence and understanding of code.

Table 1 shows some mistakes code reviewers found in 6.005 student code this semester, and queries that might locate other examples of these mistakes.

Mistakes found	Sample Query
Find examples of Board classes that assume that the size of the file input is exactly 2.	`public Board {{{ !!!(((.size() != 2))) }}}`
Find all code comments containing the word TODO, usually indicating that something was not implemented.	`///***TODO***///`
Find examples of the ConcretePage class in which the student implemented the equals or hashCode methods. ConcretePage is mutable, so it is unnecessary to define these two methods; they should be inherited from Object.	`ConcretePage {{{ equals ||| hashCode }}}`

Table 1. Common mistakes found by code reviewers and sample queries that describe the mistakes.

I evaluated the language on ten different queries. Code Search nearly perfectly retrieves relevant code chunks for well-formed queries that are specific and syntax-related, but has poor performance for ill-formed or imprecise queries. For well-formed queries, Code Search increases staff coverage of student errors compared to what it was without Code Search.

1.4 Contributions

The main contributions of this work are as follows:

1. Comment Search, a tool that improves the quality of code review comments by allowing reviewers to search through and reuse old comments that are relevant to new student code.
2. Evidence from reused comments to support the theory that many software engineering students make similar mistakes.
3. Code Search, a tool that allows code reviewers to increase their coverage of student code by finding matches to a specific pattern in code.

Chapter 2 Related Work

2.1 Existing Tools for Powergrading

As classes move to the online platform, class sizes increase dramatically while the staff size remains the same. Because the ratio between staff and students is far worse, it becomes extremely time-consuming to grade student work. One solution is to grade student code for correctness. In a software engineering class, whose goal is to teach programming style, grading only for correctness is not sufficient. Powergrading is a new approach to address this problem [2]. The idea behind powergrading is to group similar answers together and allow teachers to assign grades to groups as a whole. In their paper, Basu et al. explore powergrading short answers of social science questions. They found that this approach allows teachers to assign grades significantly more quickly without sacrificing accuracy and impartiality [3]. Although powergrading has been found to be effective for humanities classes, clustering and commenting on code provides a significant new challenge. First, code is typically much longer than short answer responses. In Caesar, students write hundreds of lines of code for each assignment, compared to the one to five word answers that powergrading handles. Second, clustering code that is stylistically and/or functionally similar is more complex than clustering natural language [4].

A code reviewing tool based on Rietveld specifically evaluates code quality at scale [5]. The tool is similar to Caesar in that students submit programming assignments and staff members provide feedback on particular lines of code. Notably, this tool stores all comments made by instructors and allows instructors to reuse comments. This allows instructors to provide consistent messages to students as well as making the feedback process faster. The tool was analyzed in a class size of 300 students, which is about 100 times smaller than a typical edX class. While reusing comments will certainly improve code review efficiency, it is not enough when scaling code review to a classroom with thousands of students.

Another tool that mass-grades computer programs is called the Automatic Coding Composition Evaluator (ACCE) which also mass-grades computer programs [6]. ACCE analyzes code for similarity and provides a visualization for clusters. This allows a user to see which cluster code belongs. However, this tool is intended to automate the grading process completely. While the intention is to allow graders to provide feedback to different clusters of submissions, the emphasis is on the machine component rather than the human component. For example, the tool automatically clusters code based on overall document distance which, in large coding assignments, likely represents general design decisions. ACCE falls short in identifying similarities in code on a smaller scale, such as method-wise or block-wise.

2.2 Clustering Code

Many computer programming assignments are open ended, which encourages a variety of correct implementations. This is especially true in a software engineering class, in which students have the freedom to design their implementations which may be hundreds of lines of code long. Despite the wide variety of possibilities, student code submissions fall into a much smaller set of unique approaches [7]. Researchers analyzed programs functionally and syntactically and organized them into clusters. It was expected that the correct answer would be the most popular. Crowdsourcing exploits this idea in using majority voting as one of its decision-making techniques [8]. Interestingly, there were also clusters of incorrect implementations. This suggests that students whose code falls in the same cluster would benefit from the same feedback. These findings support the creation of an interface that groups similar student responses to increase grading efficiency.

In order to organize code into clusters, it is important to recognize what it means for two programs to be similar enough to be clustered. It is possible to transform literals and identifiers to find identical code submissions. In fact, this is used to check for plagiarism [9]. However, code that is similar in style but not necessarily identical can also be useful to cluster. One algorithm identifies code clones at the block level using fingerprinting techniques at the statement level [4]. This algorithm allows the user to specify classification of clusters depending on the context. This will allow for graders to search for clusters containing a specific design pattern, for instance.

2.3 Code Search

As an alternative to automatically clustering code, evaluators might want to search through code for a specific error. In order to write a query that matches code rather than natural language, expert reviewers will need to learn a pattern language. Several such languages exist already. Some pattern languages match code lexically by extracting directly from source code, such as regular expression and LSME [10]. These languages are typically easy to use and have good performance, but are not good at identifying code syntax and structure at the character level. Other pattern languages match code syntactically by parsing code. There are several variants of this: JavaML [11] uses Extensive Markup Language (XML) tools to parse code, while LAPIS [12], ASTLog [13], and TAWK [14] use abstract syntax trees. In general, these languages allow for greater precision, but are harder to use and have worse performance. LAPIS seems to have the best performance without sacrificing precision. However, it has not been updated since its publication in 2002.

In order to write a new pattern language, it will be necessary to parse code into an abstract syntax tree. ANTLR [15] is one tool that does this. ANTLR first lexes code into tokens. Then, it parses code using a programming language grammar.

2.4 Caesar

Caesar is a code review tool used in MIT's Introduction to Software Engineering course. Its basic workflow is as follows:

1. Students submit their problem sets, which are then uploaded to Caesar.
2. Caesar divides problem sets into smaller chunks, generally a single file.
3. Caesar assigns each student and staff member 5-15 chunks to review.
4. Students and staff members review each chunk of code they are assigned. They can write a comment on a line or multiple lines of code, reply to comments written by other reviewers, and upvote or downvote comments.
5. At the end of the code reviewing period, students can read the feedback on their problem set submissions.

Figure 3 shows the code review interface. The main frame shows the student's code, with staff code grayed out. The comment bar is on the left. To write a new comment, a user clicks on one or more lines of code. In the figure, the user has clicked on line 3 to write a comment about it. The thumbs up/down buttons are for upvoting/downvoting comments, and mousing over a comment reveals a reply button for writing responses.

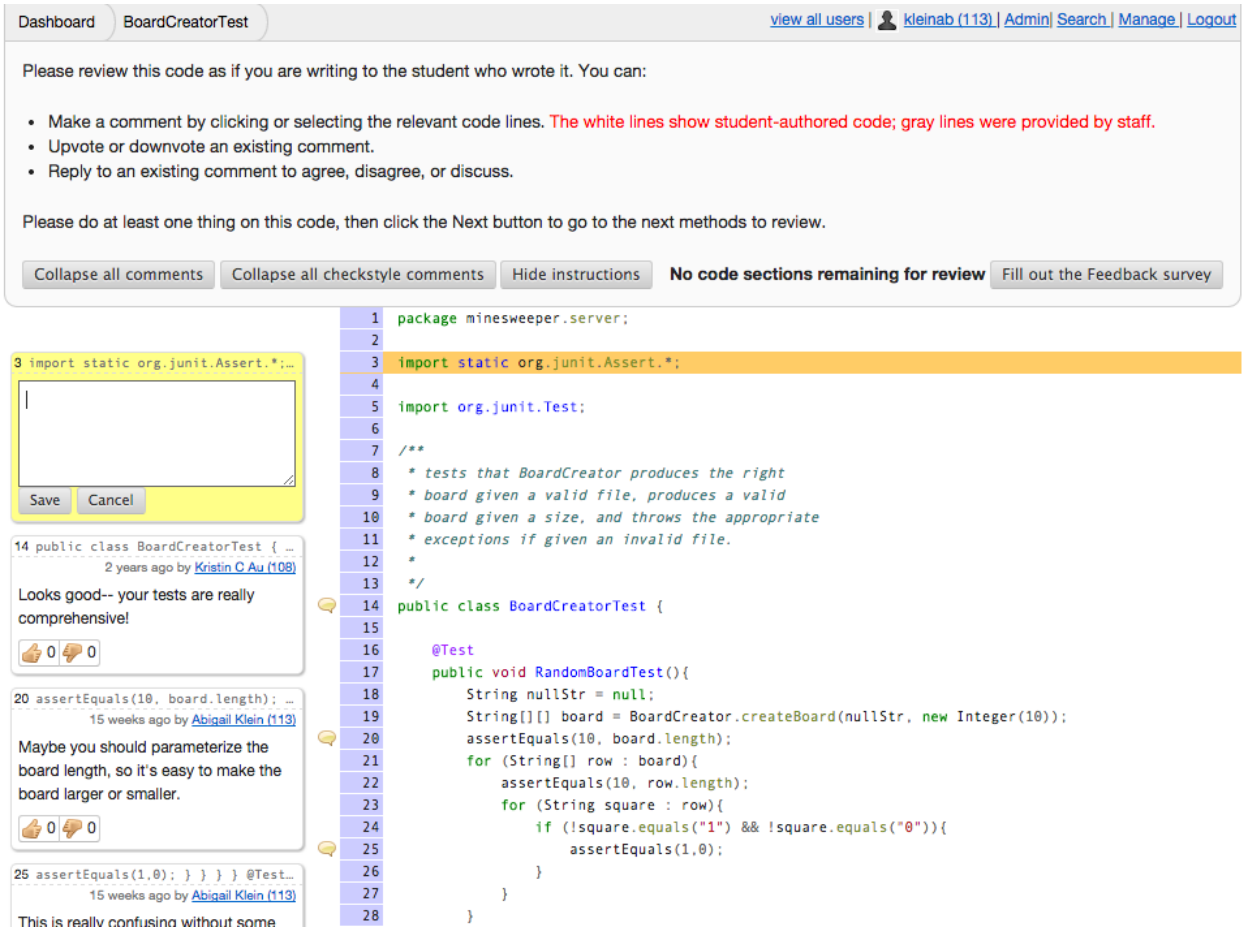

Figure 3. Caesar's code reviewing interface.

Chapter 3 Comment Search

Comment Search is a tool that allows users of Caesar to search through previous comments they have written and reuse them if they see fit. In this chapter, I describe Comment Search in detail; specifically, its motivation, user interface, implementation, and evaluation.

3.1 Motivation

The motivation behind Comment Search is to make writing comments in Caesar easier. Traditionally, students using Caesar compose a new comment for every line (or lines) of code they question. Studies show that many students make similar mistakes, so it would follow that many students write similar comments. I looked through comments written by students during one semester of 6.005. For comments that I suspected would be repeated by other code reviewers, I wrote a regular expression query, such as `magic number` or `t?mp`, to find similar comments. Some examples of the results are listed in Table 2.

Comment Description	Number of related comments	Sample comments written by students
Student should use Math.toDegrees()	81	*"magic number, could use Math.toDegrees(Math.PI)"* *"As someone mentioned in another code review, Java has a convenience method called Math.toDegrees(). And 'theta' seems a bit abstract; how about newHeading?"*
Magic numbers	209	*"Don't use "magic" numbers like 1, 3, 5, etc. Explain what these constants are."* *"Some brief comments would help to explain how the leap year is being checked. It is difficult to determine where these magic numbers come from otherwise."*
temp/tmp as variable name	40	*"name tmp var?"* *"Rename temp to something useful"*
Throw exception rather than returning 0	4	*"0 isnt a great number to return in this case, instead, throw an exception"* *"This case should never be reached, but by returning 0, it seems as if everything would be fine if your code reached this point. Consider throwing an exception, or just turning the last else if into and else."*
Test for 0^0 as input	4	*"0^0 does not meet preconditions so we do not have to test for it."* *"0^0 is supposed to be unspecified, so here it will catch a==0 first and just return 0, which works fine."*
1%m = 1	12	*"1 % m always evaluates to 1."* *"1 % m should always be equal to 1"*
Use for loop instead of while loop	6	*"A for loop is more appropriate here."* *"A for loop is more appropriate here. Also, you can just use your drawRegularPolygon below."*

Table 2. Similar comments written by Caesar code reviewers.

In total, there were 784 comments about just 21 mistakes students made in just one semester of 6.005. Presumably there are many more similar mistakes that we did not locate during our investigation. It would be useful to create a tool that makes it easier for students and staff members to reuse similar comments without needing to recompose the same comment.

Comment Search was designed for three use cases: for users who know they want to reuse a comment, users who forgot they wrote something similar before, and users who know they are writing a new comment. Users who know they want to reuse a comment can treat the text area

as a search bar and write keywords to query their previous comment. Users who forgot they wrote a similar comment before can begin to write a new comment in natural language, and Comment Search parses the sentence for keywords and suggest previous comments that contain the same keywords. Finally, because Comment Search's design is non-intrusive, users who know they are writing a new comment can ignore all of Comment Search's suggestions.

3.2 User Interface

The Comment Search interface was built on top of the existing Caesar comment interface. The existing interface looks as follows:

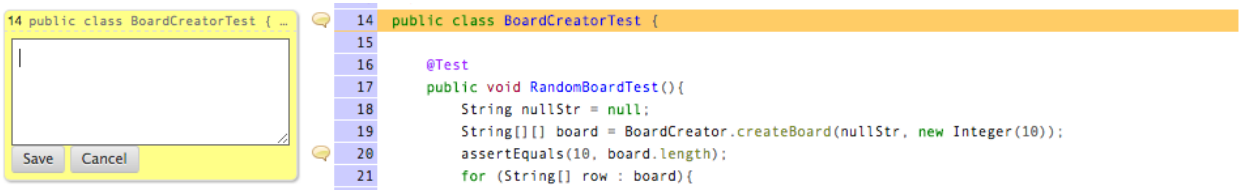

Figure 4. Caesar's comment interface.

A user clicks on one or more lines of code to write a comment. Upon clicking, the lines are highlighted and the yellow comment interface appears. The user enters a comment and presses Enter. Then, the comment is saved, which looks like this:

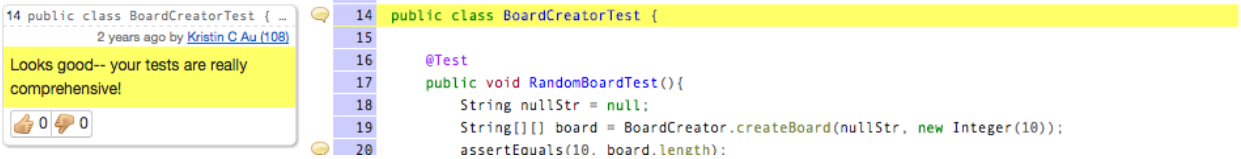

Figure 5. A newly-saved comment in Caesar.

Comment Search integrates seamlessly with the existing interface. The user sees the same yellow comment interface as in Figure 4 when trying to write a comment. As they write a comment, the system treats their unfinished comment as a search query and queries similar comments. These similar comments are loaded dynamically while the user is typing. The top three similar comments are displayed for the user to select from. This balances showing enough option to give variety while not overwhelming the user by showing too many.

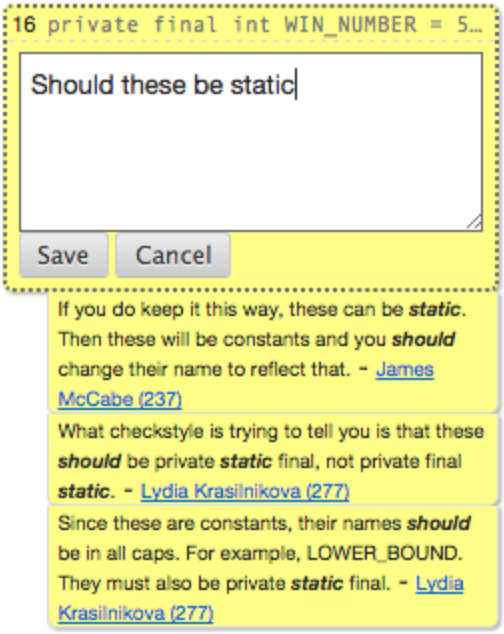

Figure 6. The Comment Search interface with three similar comments displayed.

Comment Search is non-intrusive in that a user can write their own comment and ignore the suggestions. Similar comments are listed outside of the main comment entry form so that their appearance and disappearance does not distract from comment entry.

Matching keywords are bolded. Comment Search displays the comment author and the author's point reputation, which is the number of upvotes they have received for their comments.

In order to navigate the list of similar comments, the user can use his/her mouse or tab/arrow keys. Initially, Comment Search only supported mouse events. Pilot users suggested that using arrow keys might make Comment Search more efficient and natural to use, since the user is already using his/her keyboard to type a comment. Furthermore, users are accustomed to using arrow keys when using search interfaces such as the Google search bar. In order to teach the user that they could use arrow keys, the highlighted div in the navigation list is circled by a dashed border. This is similar to the border that surrounds a highlighted cell in a spreadsheet. When a user inspects a similar comment, the user can see a context bubble showing the line or lines of code which the similar comment is referencing, as shown in Figure 7.

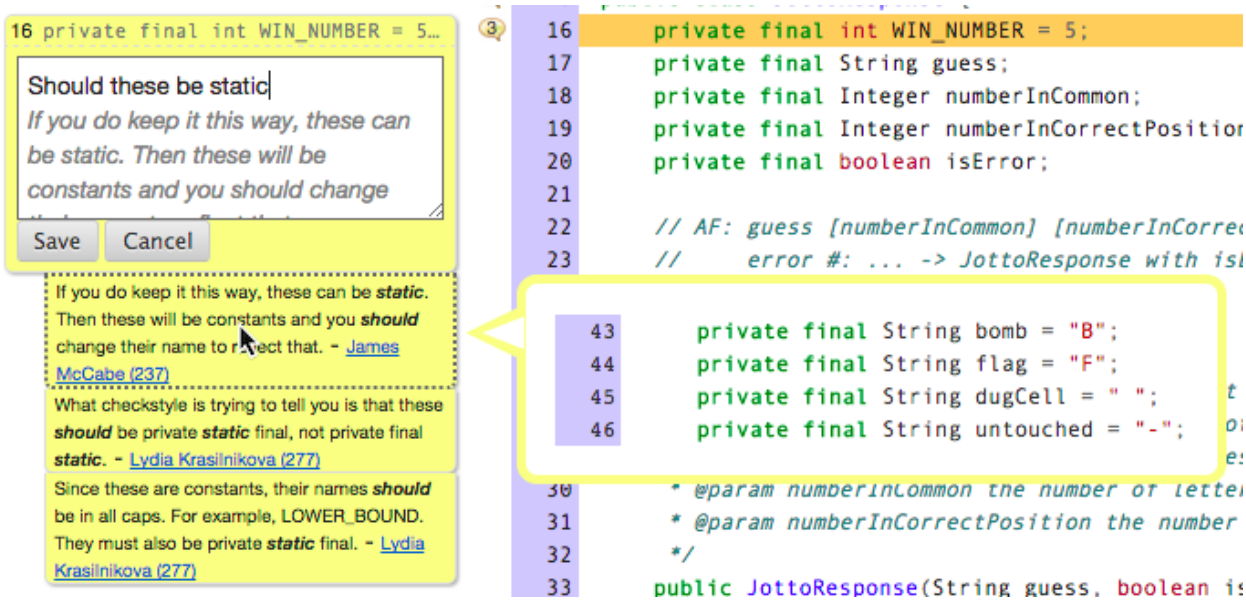

Figure 7. Comment Search context bubble. When code reviewers hover over a suggested comment, they can see a context bubble containing the lines of code that the original comment referred to. The italicized text that appears in the textbox shows where the suggested comment will appear in the textbox if the user selects it.

The user can click on this context bubble to view the entire chunk of code in a new page.

When the user inspects a similar comment, the comment is displayed in the textarea in grey and italics to demonstrate where it will go should the user select the comment. When a user selects a similar comment (either by pressing enter or by clicking it), the similar comment is appended to the bottom of the text, as shown below:

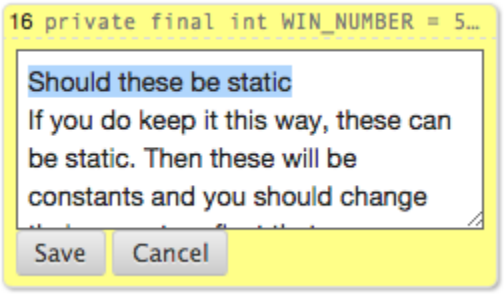

Figure 8. Comment Search text area after user selects a comment.

There are two ways a user might use a similar comment: they might want to use the entire thing, or they might want to modify it for a new context. Comment Search makes it easy for the user to do either. The user's original query is selected so it is easy to delete (or navigate to the beginning or end) if the user chooses to do so.

Comment Search is also available for a user when writing a reply and editing a comment:

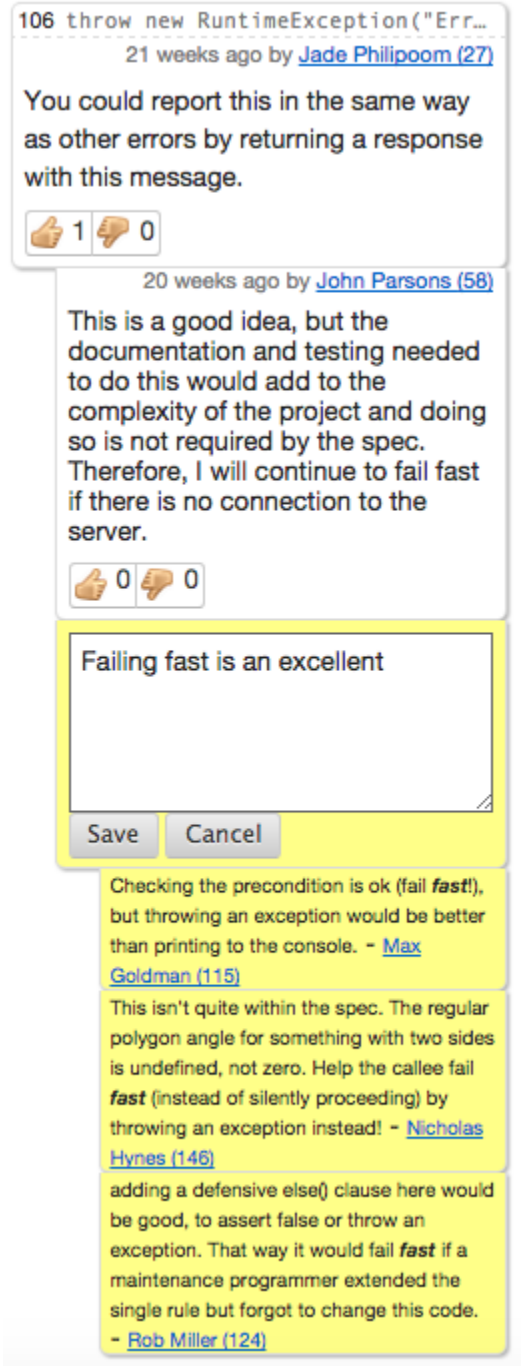

Figure 9. Comment Search for a reply.

3.3 Implementation

Comment Search is written in the Django framework using HTML, CSS, Javascript, and Python. It is a feature of Caesar. Caesar stores its data in a SQLite database.

The set of previous comments to be searched through is different for students and staff members. This distinction is necessary because students are evaluated on their ability to write comments as part of the class. Therefore, students are allowed to reuse any of their own comments so that every comment they submit is their original work. Staff members can reuse any comment written by any staff member. Most students write 200 comments over the course of one semester, and as a group, staff members write over 5000 comments in one semester.

A major design consideration was whether to search for previous comments server-side or client-side. The advantage of a server-side search is that Comment Search is faster to initialize. For staff members, it takes several minutes to load all 5000+ staff comments from the server to a client-side database, and more in places without high-speed internet. However, the advantage of a client-side search is that once the comments are loaded, searching for comments is faster. The Comment Search interface is dynamic, displaying updated search results every time the user updates the search query. In an online classroom, this would mean around 20,000 students requesting that the server perform a small search for every letter the user types into the textbox. Since code review happens over two days, this would place a high load on the server and the network. Comment Search needs to appear smooth in order for it to remain unintrusive, so it is better for it to be slow to initialize than slow to work. Therefore, Comment Search uses a client-side search engine, called Fullproof (`http://reyesr.github.io/fullproof/`). Fullproof is a client-side scoring engine written in Javascript. It stores data in an IndexedDB database in Mozilla Firefox and Web SQL for Google Chrome and Safari.

In order for the code review page to load smoothly while Comment Search loads previous comments to the client-side database, Comment Search first loads only the comments written by the user via an Ajax call. This takes around 10-20 seconds, during which time the code reviewer is most likely reading code. Then, Comment Search performs a second Ajax call to load the rest of the staff comments.

Comment Search scores each comment in the database against the query and returns a ranked list. Scores are computed based on keyword similarity to the query; the higher the score, the more relevant the comment is to the user's query. The top three highest-scoring comments are returned. These three comments must be scored above a threshold to avoid returning weak matches. This threshold was determined by reviewing comment scores and noticing the approximate score below which comments are almost always irrelevant.

In order to accommodate full English sentences as search queries, Comment Search modifies queries and previous comments. Capitalization and punctuation are ignored; Common English

words, such as “you” and “they”, are removed in order to isolate technical keywords; words are stemmed so that verb conjugations do not negatively affect the system’s performance; and duplicate letters are removed to reduce search errors from misspellings. The database stores two tables: one with all comments processed with the normal index and one with all comments processed with the stemming index. The normal index is weighted higher than the stemmed index. Figure 10 shows the post-processing pipeline.

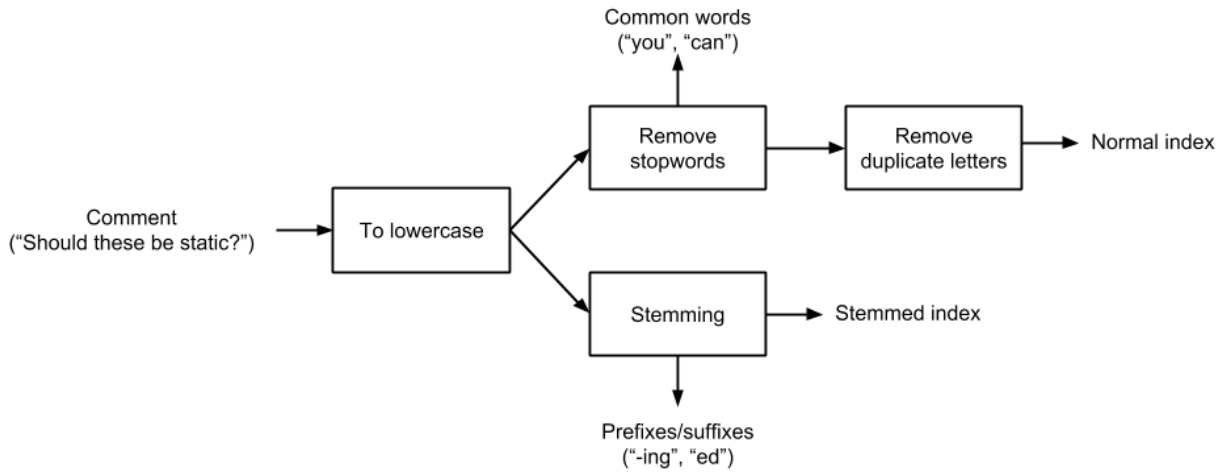

Figure 10. Data processing pipeline of Fullproof scoring engine.

3.4 Evaluation

We deployed Comment Search in Caesar to be used for a full semester of 6.005. Out of 33,483 comments written, students and staff used Comment Search to reuse 264: nearly 1%.

3.4.1 Usability Evaluation

In order to evaluate the usability of Comment Search, Comment Search logged every time the user interacted with the system. For the duration of this section, “selection” is defined as a user deciding to reuse a comment.

	Navigation	**Selection**
Keyboard	646	15
Mouse	25217	325

Table 3. Type of apparatus used to interact with the system.

Keyboard interactions were included in order to make Comment Search more efficient—it is faster to press a key than to transfer from the keyboard to a mouse or trackpad. Additionally, keyboard interactions emulate the way most people interact with search bars. The result that mouse events were more common than keyboard events suggest that the keyboard affordances

were not obvious, so users did not know they had the option of using their keyboard to interact with the system. While the system offers an efficient navigation system, it is not learnable.

Our design decision to prioritize safety over efficiency resulted in some users writing confusing comments. When a user selects a similar comment, all of the text in the text area of the comment form is highlighted and the similar comment is appended to the end, as shown in Figure 8. Comment Search highlights this text so that it can easily be deleted or modified if necessary. A few comments that people wrote contained remnants of this early search query. For example, one comment read: *"Where's your hashWhere's your hashCode and equals functions?"* This means that it was not obvious to some users that they needed to edit their comment before they saved it. However, the occasional confusing comment is a preferable result to the alternative: assuming the user wants to use only the similar comment and deleting their entire query.

Overall, the performance of the system shows that it is effective: 1% of comments were reused using Comment Search. This means that despite students and staff members being randomly assigned code to review, 1% of the time they found a repeated mistake, and Comment Search was able to help. The next section will elaborate on the successes of the system's effectiveness.

3.4.2 Effectiveness Evaluation

To evaluate the effectiveness of the system, we analyzed all of the comments written using our system. We asked two questions: How often does a code reviewer find multiple locations to reuse the same comment? And how often does a code reviewer change a previous comments when reusing it? For the remainder of this thesis, we define a comments that change upon reuse to be "evolving comments".

Figure 11 displays the results.

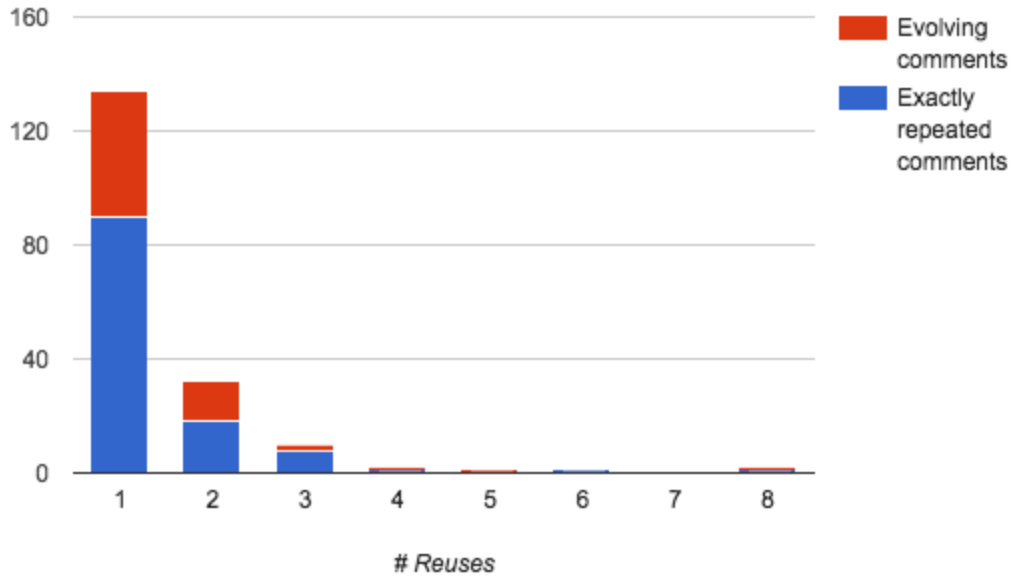

Figure 11. Chart of comment reuses using Comment Search. The x-axis is the number of times a single comment was reused: how many times a code reviewer found the same mistake. The bar chart is stacked: the blue represents the number of comments that were repeated just once and the red represents the number of comments that changed for the new context.

Out of the 264 reused comments created using Comment Search, nearly half (49%) were reused multiple times, in 48 variations. This was a surprising result: we expected many fewer variations of reused comments. Furthermore, we expected these multiple reuses to be fairly generic comments about Java programming or software engineering style, rather than specific to the problem set. However, we found a mixture:

"You should declare these private variables above"
"Reuse code and call another function you wrote which does the same thing"
"Make sure you go back and write this"
"I didn't do this either but I saw a person who has tweets that are named according to the test cases they are assigned to (eg. tweetNoFollowing, or tweetNoMentions). It might be a good idea to use those names, which are easier to understand than tweet1, tweet2...etc"
"Keeping a constant NUMBER_SIDES_IN_SQUARE and then using a call to calculateRegularPolygonAngle(NUM_SIDES_IN_SQUARE) would help to avoid the use of a magic number here."

This means that even in the randomly assigned 10 chunks to review for a problem set, students and staff found repeated mistakes, confirming that they are abundant.

The most interesting result of the system was evolving comments: comments that the user modified when reusing. Slightly over a third (36%) of comments were changed over time. We found several categories of these comments:

Grammar and spelling improvements

"You may also want to test the order at which words appear in a tweet vs the order they appear in the lis of words." → "You may also want to test the order at which words appear in a tweet vs the order they appear in the list of words."

"In your partitioning comment, say with test covers parts of the partition. This will help you determine if you've covered all the necessary cases" → "In your partitioning comment, say which test covers parts of the partition. This will help you determine if you've covered all the necessary cases"

Modify for a new application

"Consider moving these in with your tests so that all the tests for guessFollowsGraph are in one place and all the tests for influencers are in another place." → "Consider moving these in with your tests so that all the tests for writtenBy are in one place and all the tests for inTimespan are in another place, etc."

"val / 10 truncates since val and 10 are integers. Are you sure this is what you want?" → "this.getFilterValue(Filter.BLUR) / 10 truncates since both values are integers."

Elaboration

"Don't leave commented code in your commits." → "Don't leave commented code in your commits. I realize this is probably to help you while you were coding and possibly for reviewers later, but the problem with this approach is that if someone were to change the code at the top, they would very likely forget to change this which leads to trouble later."

"Instead of testing if writtenByYourself == false, you can just use if(!writtenByYourself)." → "Instead of testing if writtenByYourself == true, you can just use if(writtenByYourself).

The same applies to all the other if and else if statements. Use the ! operator to negate a boolean, when testing for == false. For example, you can write if(!writtenAsCourseWork)."

Concision

"You can replace this block with a return statement to make the code easier to read and understand." → "You can replace this block with a single return statement to simplify the code."

"Remove unused code to increase readability--or, at the very least, include a comment as to why this snippet is left in the code." → "Remove unused code to increase readability--or include a comment as to why this snippet is left in the code."

More assertive/confident

"Since these variables do not change after you initialize them, it might be better to make them final." → "Since these variables don't change once you initialize them, you should make them final."

Although staff members have the option to reuse any staff member's comments, staff members almost always used their own comments. There are several possible explanations. The first relates to the implementation. Comment Search loads all previous comments from the server in two chunks: first the individual's comments were loaded, and then all staff comments were loaded. Perhaps some staff members were so fast to write comments in comparison to the load time from the server that they weren't presented with other staff comments. However, the data shows that staff members did read and interact with other staff comments—they just didn't select them. The second explanation is that staff members remembered seeing a mistake and giving feedback on it previously, and so they searched for their previous comment specifically.

When a user saves a new comment, Comment Search saves the ID of the reused comment so that we can track the evolution of comments. There were 9 instances where Comment Search recorded that a new comment had reused a previous comment but closer analysis revealed that the two were clearly unrelated. This likely has to do with the similarity metric; two comments are similar if they share a substring of at least 20 characters. It would be better to define similarity as sharing keywords.

3.4.3 Applications for Code Search

The multitude of reused comments prompted the creation of a new tool. For many of the reused comments, the lines of code they refer to contain a similar pattern.

For example, here are two similar comments and their associated lines of code:

val / 10 truncates since val and 10 are integers. Are you sure this is what you want?	`CSSFilterFunctions.put("blur", val/10+"px");`
this.getFilterValue (Filter.BLUR) / 10 truncates since both values are integers.	`if(this.getFilterValue(filter.BLUR) !=0) cssFilterValues.put("blur", Integer.toString(this.getFilterValue(Filter.BLUR)/10) + "px");`

Table 4. Example of similar comments and their correspondingly similar lines of code.

We can imagine writing a search query that would match a division sign followed by an integer literal (one that does not contain a period). Searching for code would give staff members more control over their review assignment process. I describe this new tool, called Code Search, in the next chapter.

Chapter 4 Code Search

Code Search is a tool that finds chunks of code that match a pattern query in a large corpus of code. In this chapter, I describe Code Search in detail; specifically, its pattern language, implementation, and evaluation.

The motivation behind Code Search is to enable staff members to search for similar mistakes in code. Traditionally, students and staff are randomly assigned ten chunks of code (typically a single file) to which they provide feedback. This means that in an online classroom with approximately 20,000 students, staff members will not read much student code every week. We would like to create a tool that allows staff members to find patterns in code with the later goal (beyond the scope of this thesis) of creating a user interface for staff to give feedback to all code matches.

4.1 Pattern Language Design

We designed a pattern language to search through code. The language was designed for expert programmers. There is very minimal syntax to learn in order to reduce the learning curve of the user. It is language agnostic—the same pattern language can be used to search through code Java, Python, and any other programming language with a context-free grammar.

The pattern language behaves like a keyword grep for text documents. Users type tokens they wish to search for in code, much like a user might search through keywords in a text file. While grep returns matching lines, Code Search returns the smallest subtree that satisfies a query.

4.1.1 Primitives

The main pattern primitive is any programming language token: identifiers (`Pattern`, `compile`), keywords (`for`, `if`), literals (`10`, `"a"`), separators (`{`, `.`), and operators (`<=`, `+`). For a primitive to match a token in the code, it must be an exact textual match. Therefore, `180` and `180.0` are different primitives and will match different tokens in the code. The pattern language is case sensitive.

The second primitive is comments in code. These are expressed with the commenting operator, described in the next section. The only way to search for comments in code is to enclose the text in the commenting operator. For example, the query `TODO` will not match the code

```
//      TODO
```

but the query `///***TODO***///` would.

The converse is true as well: the query `///***System.out.println***///` would match

```
/*
 * System.out.println(expectedMentionedUsers);
 */
```

but not

```
System.out.println(followsGraph);
```

For the duration of this thesis, we use *primitive* and *query token* interchangeably. This is contrast to *code tokens*, which are tokens found in the code's abstract syntax tree.

4.1.2 Operators

The pattern language has its own operators that allow users to include logic in the query.

Type	Operator
Or	`|||`
And	`&&&`, Space
Expression Grouping	`((( )))`
Block Containment	`{{{ }}}`
Not	`!!!`
Comments in Code	`///*** ***///`

Table 5. Pattern language operators

We chose to use triple characters as operators in order to avoid namespace collision with other programming languages. The pattern language can differentiate between `&&`, which matches the *and* operator in Java code, and `&&&`, which is a logical operator in the language. Triple characters as operators are still intuitive to an expert programmer.

The conventional precedence of boolean expressions is to evaluate in this order: primitives, parentheses, not, and, or. Code Search follows this conventional precedence. Tokens and comments in code are both primitives. Additionally, Code Search introduces containment to be equal precedence as not. They are evaluated from left to right.

Or

Users can write patterns that match one expression *or* another expression. The query `tweet0 ||| tweet1` matches any block of code that contains the token tweet0 or tweet1:

```
tweet1 = new Tweet(0, “alyssa”, “is it reasonable to talk about
   rivest so much?”, d1);

private static Tweet tweet0;
```

Note: because Code Search returns the smallest subtree that matches the query, in this case Code Search would just return the identifiers `tweet1` and `tweet0`, respectively, for the two lines of code listed above. The full lines of code are included only for context.

And

Users can write patterns that match one expression *and* another expression. The query `Math.atan2 &&& 180.0` matches any block of code that contains both the tokens `Math.atan2` and `180.0`, in any order:

```
double targetAngle = java.lang.Math.atan2(targetY-currentY,
   targetX-currentX)/java.lang.Math.PI * 180.0;

double newHeading = (180.0/Math.PI)*(Math.atan2((double) targetX-
   currentX), double (targetY-currentY)));
```

Spaces between two expressions in a query are treated the same as the &&& operator. The query above is equivalent to `Math.atan2 180.0`.

Expression Grouping

Users can use parentheses to group expressions in order to change order of operations. The query `== &&& ((( false ||| true )))` matches all code that contains the operator `==` followed by either the boolean `false` or `true`:

```
if (writtenByYourself == true) {
     return true;
}
```

Block Containment

Users can specify that an expression must be fully contained by a subtree of code. The syntax is `token {{{ expression }}}`. This means that the immediate parent of the token in the code's abstract syntax tree must fully contain the expression. The query `for {{{ forward turn }}}` matches all code in which the tokens `forward` and `turn` are inside of a for loop:

```
for (int i = 0; i < 4; i++){
    turtle.forward(sideLength);
    turtle.turn(90);
}
```

Not

Users can write patterns that match subtrees that do not contain a particular expression. The query `drawSquare {{{ !!!drawRegularPolygon }}}` matches all subtrees titled `drawSquare` that do not contain any instance of `drawRegularPolygon`:

```
public static void drawSquare (Turtle turtle, int sidelength) {
    for (int side = 0; side < 4; side++){
        turtle.forward(sideLength);
        turtle.turn(90);
    }
}

public static void drawSquare (Turtle turtle, int sidelength) {
    for (int side = 0; side < 4; side++){
        turtle.forward(sideLength);
        turtle.turn(90);
    }
}
public static void drawRegularPolygon (Turtle turtle, int
  sidelength) {
    ...
}
```

The second example is a valid match because the `drawRegularPolygon` function is not contained within the `drawSquare` function. In the abstract syntax tree of this code, `drawSquare` is not one of the ancestors of `drawRegularPolygon`.

For a query in which the not character is not inside a containment, such as `!!! (((private checkRep)))` Code Search matches any file which does not contain the enclosed expression. In this case, Code Search matches any file that does not contain the tokens `private` and `checkRep`.

Comments in Code

Users can search for comments in code. The query `///***TODO***///` matches all comments that contain the word TODO:

```
/*
 * TODO specification
 * <p> PS2 instructions
 * This class must implement the required ADT interface and MUST
NOT
 * depend on photo or style representations
 */

//      TODO
```

4.1.3 Return Type

The pattern language matches the smallest subtree (block, statement, or literal) that contain all of the query words. If a file contains more than one instance of each at least one token, the file will contain multiple matches. After searching through the entire corpus of programs, each subtree is weighted based on token order, token containment, token proximity, and token distance in the abstract syntax tree. The subtrees are sorted based on their costs and returned in order. The cost calculation is described in more detail in section 4.2.

4.2 Implementation

For each file of student code, Code Search obeys the the basic pipeline displayed in Figure 12. Code Search is written using the ANTLR framework, version 4, with a Java program. We use a Python script to display the results.

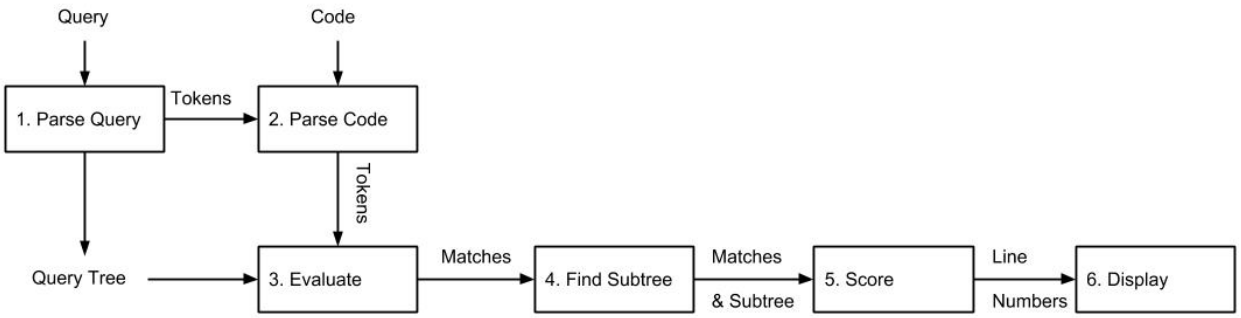

Figure 12. Code Search pipeline.

4.2.1 Query Parsing

I wrote a context-free grammar to parse queries. The grammar respects order of operations. They are applied in the following order: tokens and comments, parenthesis, containment and not, and, or.

```
OR → '|||'
AND → '&&&'
NOT → '!!!'
OPENPAREN → '((('
CLOSEPAREN → ')))'
STARTCOMMENT → '///***'
ENDCOMMENT → '***///'
OPENBRACE → '{{{'
CLOSEBRACE → '}}}'

queryUnit → exp
exp → orExp
orExp → andExp ( OR andExp )*
andExp → atomExp ( AND? atomExp )*
atomExp → parenExp | containExp | notExp | commentExp | Token
parenExp → OPENPAREN exp CLOSEPAREN
containExp → Token OPENBRACE exp CLOSEBRACE
notExp → NOT atomExp
commentExp → STARTCOMMENT .*? ENDCOMMENT
```

Figure 13. Context-free grammar for Code Search pattern language.

A Token is any integer or floating point number, any string, or the programming language's special characters. For Java, these are separators (`{`, `.`) and operators (`=`, `&&`). The grammar skips whitespace. Spaces and `&&&` are both operators for *and* expressions and mean the same thing.

The user supplies a query; for example, `calculateRegPolyAngle {{{ 2 }}}`. ANTLR parses this query into a tree. Code Search keeps track of all of the primitives (which are the leaf nodes of the query tree) for the next step.

4.2.2 Code Parsing

Code Search parses each file of student code into an abstract syntax tree. During this process, Code Search creates a mapping of all tokens from the query to primitives (Tokens and comments) in the code whose text matches.

Comments are by default skipped by programming language grammars. Code Search modifies the grammars to send them to a hidden channel, which it can access during code parsing. This way, users can search through comments in code as well.

Code Search is language agnostic: we include grammars for Java 7, Java 8, and Python. Code Search attempts to parse the file using each grammar, and moves on to the next one in order should the parsing return errors. We include both Java 7 and Java 8 because parsing a file using the Java 8 grammar supplied by ANTLR is approximately 20 times slower than using the Java 7 grammar. Most files can be parsed using Java 7; only those that use new features, such as lambdas, require Java 8.

4.2.3 Evaluation

Code Search treats primitives as boolean operators when matching a query against a code subtree. Each primitive is associated with a `True/False` value, referring to whether that primitive is or is not supposed to be found in the resulting code subtree, based on the query. For example, the primitive `while` would have a positive (`True`) value for the query `while` and a negative (`False`) value for the query `!!!while`.

To evaluate a query, Code Search recursively evaluates every node of the query tree using the tokens from the parsed code. We think of each result of Code Search as a boolean expression in conjunctive normal form (CNF): a conjunction of clauses, where each clause is the disjunction of primitives. A clause is a list of primitives that satisfies a query and their associated `True/False` value. Representing each Code Search solution in this way allows us to treat the query as a Boolean expression and solve it with the classic definitions of *And*, *Or*, and *Not*.

Figure 14 contains pseudocode for matching a query against the code tree.

```
eval(query subtree, code tree):
     switch (type of tree):
          case Query Unit:
               // Query Unit is in the form: child
               eval(child, code tree)
               if any token is missing from the code and its boolean
                  value is True: // the query requires it
                     remove that clause from the list of solutions
               if any token exists in the code and its boolean value
                  is False: // the query requires that it not exist
                     remove that clause from the list of solutions
               return all remaining solutions
          case Or:
               // Or is in the form: child ||| child ||| child...
               result = []
               for each child in or statement:
```

```
                result += eval(child, code tree)
          return result
    case And:
          // And is in the form: child &&& child &&& child...
          result = []
          for each child in and statement:
                result += eval(child, code tree)
          return cartesianProduct(result)
    case Atom:
          // Atom is in the form: child
          return eval(child, code tree)
    case Paren:
          // Paren is in the form: ((( child )))
          return eval(child, code tree)
    case Contain:
          // Contain is in the form: Token {{{ child }}}
          eval(child, code tree)
          for each result in child eval:
                construct a list of common ancestors // all
                   subtrees that contain all of the positive
                   Tokens and none of the negative Tokens in the
                   clause
                if Token is not one of the common ancestors:
                      remove this clause from the list of
                         solutions
          return all remaining solutions
    case Not:
          // Not is in the form: !!! child
          // In order to negate the entire CNF, we perform
             DeMorgan's law
          result = eval(child, code tree)
          for each primitive in result:
                toggle boolean value
          return cartesianProduct(result with toggled booleans)
    case Comment:
          // Comment is in the form: ///*** text ***///
          if text is empty:
                return a list of all comments in the code
          if code contains comment text:
```

```
                    return [{code token nearest Comment: True}]
               else:
                    return [{empty token: True}]
          case Token:
               if code tree contains token:
                    return [{code token: True}]
               else:
                    return [{empty token: True}]

function cartesianProduct(sets):
    Take the cartesian product of a set of sets to return all
       possible permutations
    ex. sets is [[[A, B], [C, D]]]
        return  [[A, C], [A, D], [B, C], [B, D]]
```

Figure 14. Pseudocode for evaluation of Code Search.

The evaluation function returns every possible permutation of results in the code that matches the query.

Code Tree then computes the smallest complete block of code that contains all of the positive code tokens in a result set and doesn't contain any of the negative code tokens. This is returned as the smallest common subtree in the code.

4.2.4 Scoring

We score results using four metrics: ordering cost, containment cost, adjacency cost, and distance cost. Each cost is a floating point value between 0 and 1, with low scores given to strong candidates.

Ordering Cost

Results that match the query in order should have a lower cost than results whose tokens are out of order in the code. Code Search returns a score of 0 if all positive tokens are in order based on their location in the document and 1 otherwise.

Containment Cost

A *token's subtree* is defined to be the subtree of the token's immediate parent in the code's abstract syntax tree. Containment means that a token's subtree contains another token. Results that have containment should have a lower cost than results whose tokens are all siblings. Note that if a user uses the containment characters in their query (`{{{` and `}}}`), containment is required.

To calculate containment cost, first we calculate the *number of token containments*: the number of tokens that are contained inside another token's subtree. As an example, several students received the comment *"A return statement breaks out of the function, so the use of else in not necessary."* To search for patterns like this, a staff member might write the query `if return else`. This could match several structures of code:

```
if (writtenByYourself == false) {
     if (availableToOthers == false) {
          return false;
     }
     return true;
}
else {
     return true;
}
```

None of the tokens are contained inside any other token subtrees, so the number of containments is 0.

```
if (writtenByYourself == true) {
     return true;
}
else {
     ...
}
```

The `return` token is contained inside of the `if` token's subtree, and the `else` token is not contained inside any token's subtree. The number of containments is 1.

```
if (writtenByYourself == false) {
     if (availableToOthers == false) {
          return false;
     else {
          return true;
     }
}
```

The `return` token and the `else` token are each contained inside of the `if` token's subtree, so the number of containments is 2.

```
if (writtenByYourself == false) {
     if (availableToOthers == false) {
          return false;
     else if (writtenAsCoursework == true) {
          return true;
     }
}
```

The `return` token is contained inside the `if` token's subtree and the `else` token's subtree. The `else` token is contained inside the `if` token's subtree. The number of containments is 3.

In order to normalize the cost between 0 and 1, we divide by the maximum number of containments for a given query, which is $\frac{n(n+1)}{2}$ where $n+1$ is the number of tokens in the query. So, the containment cost cost is $1 - \frac{number\ of\ containments}{maximum\ number\ of\ containments}$.

Adjacency Cost

Results with many tokens that are right next to each other should have a lower cost than results with tokens that are far away from each other. Adjacency cost calculates the cumulative closeness of every token in the result set based on how many tokens are between them. In order to normalize this cost, we divide by the farthest possible distance between two tokens, i.e. the total number of tokens in the file. Importantly, the weighting should not be linear. Two tokens that are 100 tokens apart are about as bad as two tokens that are 150 tokens apart. However two tokens that are 2 tokens apart are significantly more desirable than two nodes that are 52 tokens apart. A logarithmic scale is more appropriate. Thus, the adjacency cost is $\frac{log(\sum_{tokens\ in\ query} number\ of\ tokens\ between\ two\ tokens)}{log(number\ of\ tokens\ in\ file)}$.

Distance Cost

Results with tokens that are close to each other in the coding structure should have a low cost. For example, if we want to search for two while loops right after each other, it should not matter how large the while loops are. The distance cost measures the length of the path in the code tree between two tokens. It is normalized by the maximum path length, which is twice the depth of the tree. Thus, the distance cost is $\sum_{tokens\ in\ query} \frac{path\ length\ between\ two\ tokens}{maximum\ path\ length}$.

The costs are then weighted to return a final cost used for ordering the results. Currently, the costs are weighted as follows:

$$\begin{aligned} weightedCost = {} & 0.4 \times orderingCost \\ & + 0.1 \times containmentCost \\ & + 0.25 \times adjacencyCost \\ & + 0.25 \times distanceCost \end{aligned}$$

These values were chosen empirically. We anticipate determining a proper weighting for the costs based on which results are most relevant to users.

4.2.5 Display

Once all results are scored, we order them from lowest cost to highest cost and display them all. We highlight the tokens selected in the result. Figure 15 displays the top 4 results from a query that matches while loops that should be for loops.

```
private-sp15-ps0-beta-0-day-[username]-src-turtle-TurtleSoup.java,
cost: 0.0980489985411
        while(i<sides){
            i+=1;
            turtle.forward(sideLength);
            turtle.turn(180 - calculateRegularPolygonAngle(sides));
        }

private-sp15-ps0-beta-0-day-[username]-src-turtle-TurtleSoup.java,
cost: 0.11197234122
        while(adjustment<0){
            adjustment += THREE_SIXTY_DEGREES;
        }

private-sp15-ps0-beta-0-day-[username]-src-turtle-TurtleSoup.java,
cost: 0.11763198412
        while (headingDiff < 0) { headingDiff += 360d; }

private-sp15-ps0-beta-0-day-[username]-src-turtle-TurtleSoup.java,
cost: 0.118557352166
        while (i<NUMBER_STARS){
            i++;
            drawStar(turtle, sideLength);
            turtle.turn(STAR_TURN_ANGLE);
            //calculate the side length of the next largest star
            sideLength=(int)
              Math.round(sideLength*Math.pow(GOLDEN_RATIO, 3.0)
                                /Math.sqrt(GOLDEN_RATIO+1));
            //next three lines bring the turtle to a spot to begin
              the next star
            stepSize=(int)Math.round(sideLength*Math.pow(GOLDEN_RATIO
              , -2.0));
            turtle.forward(stepSize);
            turtle.turn(TURN_AROUND_ANGLE);
        }
```

Figure 15. Code Search results. Each result shows the filename, cost, and block of code, with tokens highlighted.

4.3 Evaluation

I evaluated Code Search for its major use case: a teaching assistant for 6.005x who wishes to search for student code that matches a certain pattern. The teaching assistant begins by writing a query. Code Search returns a list of all blocks of student code that match the query pattern, ranked in order of relevance. The teaching assistant then scans the list of code blocks and selects the pertinent blocks. Most likely, after no more than a few minutes of scanning, the teaching assistant will lose patience or interest. We estimate that the teacher can scan through 50 Code Search results in a few minutes, so our evaluation only analyzes the top 50 results. If the query does not match enough relevant code, the teaching assistant may try to modify the query to get more hits.

4.3.1 Qualitative Evaluation

I evaluated Code Search in two ways: information retrieval accuracy and performance. To do this, I collected 273 examples of reused comments (from Comment Search) from a semester of 6.005. I analyzed the lines of code that these comments addressed and wrote search queries for Code Search that would find these lines. I ran these queries on the entire corpus of 6.005 files from that problem set to see how many other examples of this mistake there were.

57 search queries covered 98 of the 273 reused comments (36%). 41 of the queries were repeated: the same query could cover multiple comments due to repetitions and evolutions of comments from Comment Search. Interestingly, several of the repeated queries were for comments written by multiple people, supporting the need for the Code Search tool. Section 4.3.4 elaborates on the 175 reused comments which could not be described with a Code Search query.

For testing, I focused on the first problem set of 6.005, in which students are still adjusting to writing Java for the first time. Many reused comments were about semantic errors, which are easy to capture with Code Search. Some examples are listed on the following pages.

Comment	*"You could call drawRegularPolygon in this method to keep the code DRY [Don't Repeat Yourself]. This loop is repeated in drawRegularPolygon."*
Query	`drawSquare {{{ !!!drawRegularPolygon { } }}}`
Relevant responses	`public static void drawSquare(Turtle turtle, int sideLength) { final double TURN_ANGLE = 90.0; turtle.forward(sideLength); turtle.turn(TURN_ANGLE); turtle.forward(sideLength); turtle.turn(TURN_ANGLE); turtle.forward(sideLength); turtle.turn(TURN_ANGLE); turtle.forward(sideLength); turtle.turn(TURN_ANGLE); }`  `public static void drawSquare(Turtle turtle, int sideLength) { int NUMBER_OF_SIDES = 4; for(int i = 0; i < NUMBER_OF_SIDES; i++) { turtle.forward(sideLength); turtle.turn(RIGHT_ANGLE); } }`
Redundant responses	`public static void drawSquare(Turtle turtle, int sideLength) { int NUMBER_OF_SIDES = 4; for(int i = 0; i < NUMBER_OF_SIDES; i++) { turtle.forward(sideLength); turtle.turn(RIGHT_ANGLE); } }`

Table 6. Code Search results for repeated code.

Of the top 50 results from this search, 45 were relevant. The other ones were all redundant due to various combinations of parentheses matching. The single curly braces in the query are necessary to avoid matching code such as `drawSquare(turtle, 40);` This code does not contain `drawRegularPolygon`, but it also is not the targeted mistake.

Comment	*"why not a for loop? for(int i=0; i<sides; i++)"*
Query	`while {{{ (((++|||+=))) !!!(((360|||360.|||360.0))) }}}`

Relevant responses	`while(i<sides){`
	`  i+=1;`
	`  turtle.forward(sideLength);`
	`  turtle.turn(180 - calculateRegularPolygonAngle(sides));`
	`}`
	
	`while(count<4){`
	`  turtle.forward(sideLength);`
	`  turtle.turn(angle);`
	`  count++;`
	`}`

Irrelevant responses	`while(turnAngle<0d||turnAngle>360d){`
	`  turnAngle+=((turnAngle<0d)?1d:-1d)*360d;`
	`}`

Table 7. Code Search results for while loops that should be for loops.

In this case, I wrote a query to match while loops that should be for loops. I first wrote the query as `while {{{ (((++|||+=))) }}}` which is more intuitive. However, this ended up matching many chunks of code that used a while loop appropriately, such as the irrelevant response in the table above. It turned out that for the specific problem set I tested, there was a common case involving angle adjustments which did necessitate a while loop with a counter. For this reason, this query excludes while loops that contain the number 360, which targets this query for this specific problem set. 43 out of the top 50 results were relevant. The irrelevant ones were for while loops that did not need to become for loops, such as the example above.

Comment	*"The specs already dictate that sides must be >2 so I'm not sure if this is necessary."*
Query	`calculateRegularPolygonAngle {{{ if (sides <= 2) }}}`

Relevant responses	`public static double calculateRegularPolygonAngle(int sides) {`
	`  if(sides <= 2){`
	`  //check if sides is greater than 2.`
	`  throw new RuntimeException("sides must be > 2");`
	`  }else{`
	`  return (sides - 2) * 180.0d / sides;`
	`  }`
	`}`
	
	`public static double calculateRegularPolygonAngle(int sides) {`
	`  if (sides <= 2) {`
	`  return -1.0;`
	`  /*throw new IndexOutOfBoundsException();//is it better to have some sort of error message?*/`
	`  }`
	
	`  return 180.0*(sides-2.0)/sides;`
	`}`

Redundant responses	`public static double calculateRegularPolygonAngle(int sides) {`
	`  if(sides <= 2){`
	`  //check if sides is greater than 2.`
	`  throw new RuntimeException("sides must be > 2");`
	`  }else{`
	`  return (sides - 2) * 180.0d / sides;`
	`  }`
	`}`

Table 8. Code Search results for misunderstood method specifications.

In this case, there were only 7 total relevant responses. All of the rest were redundant due to various combinations of parenthesis matching. It is worth noting that all 7 relevant responses were in the top 9 returned responses, because of the way we perform our ranking.

Comment	*"possible bug here. if heading is a very negative number, say - 720, then the if statement will only increment it to -360. To fix this, you should put a while loop (while heading <0)"*
Query	`calculateHeadingToPoint {{{ if {{{ < 0 + (((360|||360.0))) }}} }}}`
Relevant responses	`public static double calculateHeadingToPoint(double currentHeading, int currentX, int currentY, int targetX, int targetY) { double HeadingCartesian = (-1*currentHeading + 90); /* * use same coordinate system as given by atan2 */ if (HeadingCartesian < 0){ HeadingCartesian = HeadingCartesian + 360; } double angle = HeadingCartesian - (Math.atan2(targetY-currentY, targetX-currentX) * 180/Math.PI); if (angle < 0){ angle = angle + 360; } return angle; }`
Irrelevant responses	`public static double calculateHeadingToPoint(double currentHeading, int currentX, int currentY, int targetX, int targetY) { double HeadingCartesian = (-1*currentHeading + 90); /* * use same coordinate system as given by atan2 */ if (HeadingCartesian < 0){ HeadingCartesian = HeadingCartesian + 360; } double angle = HeadingCartesian - (Math.atan2(targetY-currentY, targetX-currentX) * 180/Math.PI); if (angle < 0){ angle = angle + 360; } return angle; }`

Table 9. Code Search results for a bug in angle calculation.

In this case, the query appears to be written out of order (it is more intuitive to write `calculateHeadingToPoint {{{ if (< 0) {{{ + (((360|||360.0))) }}} }}}`. But when ANTLR parses an if statement into an abstract syntax tree, the arguments to the if statement are a subtree of the if statement itself. Therefore the query needs to match this structure.

4.3.2 Information Retrieval Accuracy

We wish to evaluate the information retrieval accuracy for the use case described above. We model the teacher's patience by assuming that the teaching assistant will look through no more than 50 blocks of student code. Streaks of relevant code blocks will take less energy and time to filter through, so if there are more than 50 code blocks in a row, we assume that the teaching assistant will continue to look until he/she finds an irrelevant chunk.

The two metrics typically used to evaluate information retrieval accuracy are precision and recall. Precision is the fraction of documents retrieved that are relevant. To calculate precision, I counted the number of relevant results in the top 50 returned results by Code Search for 10 sample queries. This serves to analyze the ranking of Code Search's results, because relevant but poorly ranked results are not considered.

Student Mistake	Query	Precision @ 50
"You could use Math.toDegrees here to convert angleToTarget from radians to degrees."	`(((180 ||| 180.0))) Math.PI`	1.0
"possible bug here. if heading is a very negative number, say - 720, then the if statement will only increment it to -360. To fix this, you should put a while loop (while heading <0)"	`calculateHeadingToPoint {{{ if {{{ < 0 + 360 }}} }}}`	0.52
"The specs already dictate that sides must be >2 so I'm not sure if this is necessary."	`calculateRegularPolygonAngle {{{ if (sides <= 2) }}}`	1.0
"It would be good to include a comment on why you subtract calculateRegularPolygonAngle from 180."	`drawRegularPolygon {{{ !!!///******/// (((180|||180.0))) - }}}`	0.0
"You could call drawRegularPolygon in this method to keep the code DRY. This loop is repeated in drawRegularPolygon."	`drawSquare {{{ !!!drawRegularPolygon {} }}}`	0.90

"Redundant import from the java.lang package - java.lang.Math."	`import java.lang`	1.0
"Maybe document why you're multiplying by 180. Specifying Math.atan2(y,x)'s return range would also do- also future programmers who may want to modify your code wouldn't have to look it up."	`///*** ***/// Math.atan2 180.0`	0.06
"You can just return the operation instead of creating a new variable. The @return specification already describes this line."	`return ;`	0.08
"Keeping a constant NUMBER_SIDES_IN_SQUARE and then using a call to calculateRegularPolygonAngle(NUM_SIDES_IN_SQUARE) would help to avoid the use of a magic number here."	`turn 90`	1.0
"instead of initializing i = 0 and using a while loop, you could try implementing using a for loop instead"	`while {{{ (((++|||+=))) !!!(((360|||360.0|||360.)) ) }}}`	0.86

Table 10. Precision values for 10 sample queries.

By looking at the table, it is clear that some queries return nearly entirely relevant results, and others return almost no relevant results. This implies that Code Search is really only useful for certain types of queries. This is discussed further in chapter 5.

Recall is the fraction of relevant documents that were retrieved. Calculating the total number of relevant documents is difficult. When studies have evaluated search engines for recall, they have used a combination of multiple other search engines as a way to count the total number of relevant documents in the space. In this case, there are no other search engines, so this process would need to be done manually.

Code Search improves staff member coverage of student code. Table 11. shows the coverage ratio for the five Code Search queries that had high precision from Table 10. The number of comments written and example found were for one problem set of one semester of 6.005.

Student Mistake	Number of comments written by students and staff in Caesar	Number of examples found by Code Search (in the top 50)	Coverage Ratio
"You could use Math.toDegrees here to convert angleToTarget from radians to degrees."	46	50	1.09
"The specs already dictate that sides must be >2 so I'm not sure if this is necessary."	7	7	1.00
"You could call drawRegularPolygon in this method to keep the code DRY. This loop is repeated in drawRegularPolygon."	11	45	4.09
"Redundant import from the java.lang package - java.lang.Math."	61	45	0.73
"instead of initializing i = 0 and using a while loop, you could try implementing using a for loop instead"	9	43	4.78

Table 11. Coverage ratios of code review with Code Search compared to without.

4.3.3 Performance

Because the searching happens online with a user, Code Search needs to be very fast. At worst, a user would be willing to wait 10-15 seconds, but even this is pushing the limit. Currently, Code Search is nowhere near fast enough. Fast queries (ones with small query trees and few matching tokens in the code, such as

```
drawRegularPolygon {{{ !!!///******/// (((180|||180.0))) - }}}
```

take around 50 minutes, while some of the slower ones take well over several hours. The most complex queries, such as

```
calculateHeadings {{{ calculateHeadingToPoint(
   currentX|||xcoords.get(i), currentY|||ycoords.get(i),
   targetX|||xcoords.get(i+1), targetY|||ycoords.get(i+1)) }}}
```

actually run out of stack space and cannot complete.

4.3.4 Limitations of Code Search

For the 175 reused comments that could not be expressed using a Code Search search query, there were several common reasons why Code Search would not be useful.

Want to specify size of comment or size of line of code

"Make sure to go back and write this"

```
private void checkRep() {
}
```

"This is way too long of a method - generally cap method to 40 lines or so. Breaking it up will make your code more clear and readable."

Need to have an understanding of variables and what they mean

"You should not need to synchronize this method. You don't want to sprinkle synchronized without care."

"Could this be written more simply as a division of 100 by the size of a row?"

```
double width = 1.0/row.size();
width = (double)Math.round(width * 1000) / 10;
```

Want to use a regular expression to identify a variable or string

"You don't seem to vary the capitalization of the usernames at all - how are you testing that "alyssa" == "ALYssa"?"

"Since these are constants, their names should be in all caps. For example, ERROR_1_STRING."

"Math.atan2() takes arguments in the order: (deltaY, deltaX). I think you are doing it backwards."

Chapter 5 Discussion

In this chapter I will discuss some of the remaining questions I have not answered about Comment Search and Code Search. Finally, in this chapter I will tie together the entire Comment Search/Code Search system and explain how we foresee them to be used together in the future.

5.1 Comment Search

In my introduction, I hypothesized that Comment Search would produce higher quality comments. Our results from Comment Search support this. We found reused comments with grammar and spelling improvements, concision, elaboration, and assertiveness: all positive qualities. This means that Comment Search not only benefits code reviewers, who don't need to compose every comment anew, but the reviewees as well.

5.2 Code Search

In the evaluation of Code Search, I show that Code Search is nearly perfect at finding relevant matches for some kinds of queries, and abysmal at finding relevant matches for other kinds of queries. Code Search can find code that contains a specific pattern and whose tokens are in order very successfully. Examples of this are finding while loops that should be for loops, finding code that writes its own `Math.toDegrees()` function rather than using the Java one, and finding a frequently repeated full line of code (`if (size <= 2)`). When there is uncertainty in the query, such as an uncertain variable name or an uncertain order of tokens, Code Search returns mostly irrelevant hits in the top 50 results. The relevant hits are sprinkled elsewhere in the list of sometimes hundreds of thousands of results. And if a teacher wishes to search for an uncertain variable name, such as any variable that is capitalized, then he/she cannot write any query to do so. Naturally, as we find more of these limitations, we will modify Code Search's pattern language to accommodate as many queries as we can. I discuss some of the most pressing modifications in the next chapter.

There are two additional applications we foresee of Code Search. The first one is to reuse successful queries for future code submissions. After Code Search runs newly submitted problem sets against a bank of queries, it will produce a list of matching code and an associated comment to fix the error. A teaching assistant might need to verify that the matches are relevant to the comment before passing the comment along to the students. Or, the verifier might be removed from the process and Code Search will offer comments to students as suggestions. Regardless, we would like to keep good queries written for Code Search as a bank of complex patterns so that teaching assistants can reuse their work of creating queries.

The second application of Code Search is to be used in a non-educational setting. Code Search's pattern language is certainly useful for code reviews in an online classroom, as outlined in this thesis, but it does not need to be limited to this application. It could be helpful for searching through large software projects in a number of other settings, such as version control and industry code reviews.

5.3 The Comment Search/Code Search System

Comment Search and Code Search each contribute to staff members having greater coverage of student code for an online classroom. Together, they are stronger than the sum of their parts. In Section 3.3.3, I describe how Comment Search illustrated the need for Code Search: we realized we could search directly for the code to which the reused comments referred. Similarly, Comment Search can give staff members an idea of which mistakes are common. This way, they know which code patterns are worth the effort of writing a Code Search query.

While Code Search can certainly affect more students at once, Comment Search makes up for Code Search's limitations. There are many mistakes made in code that Code Search cannot capture because they require an understanding of the code. For example, to understand whether a program is thread-safe, a reviewer needs to have an understanding of how several methods, and possibly even classes, relate to each other. Since there are a lot of different parts to consider, it would be difficult to write a query that captures a common multithreading strategy. As another example, to grade a method specification, a reviewer needs to read and understand the comment in code. Code Search can help reviewers find a given method specification, but it would be difficult to write a query to find all poorly written specifications. Fortunately, code reviewers can still write comments about these more complicated errors using Caesar's traditional code reviewing interface and Comment Search.

Code Search is better suited for localized or syntactic mistakes—the "easy" mistakes found by students. If Code Search automatically marks many of these mistakes, then student reviewers will be forced to read code more carefully to make more complex critiques. This unintended consequence of using the Comment Search/Code Search system may actually have a positive effect on the quality of student reviewers.

Chapter 6 Future Work and Conclusion

6.1 Future work

6.1.1 Comment Search

Usability Improvements

There are a number of small improvements to Comment Search which would improve its usability. We would like to order similar comments by date: users are more likely to reuse a comment written more recently. We would like to display how many times a comment has been reused. This additional information may provide insight for teaching assistants who want to write a Code Search query about a highly repeated mistake.

Future Applications

Based on which lines of code on which users repeatedly comment, we can learn which patterns in the code may call for that comment. We would like to suggest similar comments earlier in the process of writing a comment: not just when a user begins to type, but when a user highlights a piece of code.

6.1.2 Code Search

Information Retrieval Optimizations

It is critical that Code Search returns an optimal ordering of results in order for it to work. We selected a weighting for the four scoring metrics (ordering, containment, adjacency, and distance) based on empirical testing of a limited number of test cases. It would be better to select weights based on real data from teaching assistants. We intend to use machine learning to adjust the weights to which results are most relevant to teaching assistants.

In addition, Code Search is only useful for limited types of patterns: ones that are specific and related to syntax. We would like to expand the types of patterns Code Search for which might be used. One way to do this would be to allow users to specify query tokens with regular expressions. This would resolve the problems outlined in section 4.3.4. This would mean users will be able to write queries with a generic or filler variable (`[A-Za-z0-9_]*`), which will mean queries do not have to be so specific. Introducing regular expressions would also resolve the pain in writing numbers in queries. Code Search matches literals perfectly: 360, 360.0, 360., and 360d are all different literals. Users need to write cluttered queries like `360|||360.0|||360.|||360d` in order to express the number 360.

Finally, Code Search cannot be used at all if the student code cannot compile. In the future, we will need to construct an abstract syntax tree of uncompilable code. ANTLR does not allow this, so we may attempt to compile smaller sections of the student code that can be compiled.

Performance Optimizations

There are several ways to improve performance. The main change to drastically improve performance will be to parse all student code ahead of time. Parsing all student code for a single day's submission of a problem set takes over 30 minutes. Clearly, Code Search needs to parse the problem sets ahead of time and index the data.

We can perform pruning of our massive set of results to further improve performance. Code Search was designed to return all matches to the query, and sort them by relevance, similarly to common search engines. However, it turns out that Code Search returns far too many matches than is necessary. For example, we ran the query

```
calculateRegularPolygonAngle {{{ if (sides <= 2) }}}
```

on all files of a single submission of a problem set. The query found 7 files that matched, but 1027 matches total. Most of these matches were various combinations of parentheses, as shown in Table 8—most large Java files have nearly 100 parentheses. We can eliminate many of these excess matches by eliminating any result whose score is significantly worse than the current best score. In addition, Code Search sometimes returns 10-20 matches for a single block of code within a single file, with different combinations of tokens inside. We should return only the best result from a single block of code.

Usability Improvements

Once Code Search's performance improves, we will perform a user test to evaluate Code Search's pattern matching language. We anticipate making three usability improvements here. One non-intuitive requirement of the Code Search pattern language was illustrated in Table 8: sometimes user is required to understand the structure of the code's abstract syntax tree to write a correct query. We would modify the grammar and the evaluation of Code Search in order to reduce this. A second flaw in the language is the use of triple characters to denote logical operators, such as `&&&` and `!!!`. We anticipate selecting a new set of operators that make the queries look less cluttered. Lastly, we might choose to make `&&&` and spaces mean different things to empower users to write more specific queries: perhaps `a1 a2` should mean `a1` followed by `a2`, while `a1 &&& a2` should mean `a1` and `a2` in either order.

Finally, Code Search will not be usable until it has a user interface. The user interface will be composed of three components: a query bar, a display area, and a comments box. The teacher can write their query in the query bar, aided by highlighting and parenthesis/brace matching to guide the user to write legal queries. The teacher can read matching student code in the display area. The display area will show only the relevant lines of code and highlight tokens so it can be easily read. The display area may only show 10 hits per page, but the teacher can continue to select relevant blocks of code on subsequent pages. The teacher will select code blocks to comment on, and write a comment in the comment bar. The teacher can also write a different comment for different selections if desired.

Future Applications

We anticipate using the queries written for Code Search to build a database of rules for student code. Using Code Search, these rules can be stronger than the simple ones written by checkstyle, which check for unused imports and magic numbers.

6.2 Conclusion

In this thesis, I set out to answer the question "How can software engineering teachers give more qualitative feedback to their students in an online classroom setting?" We developed two tools that work together to support teachers in this endeavor. Comment Search allows teachers to rapidly review student code. It also provides insight into common mistakes and stylistic patterns by students. Code Search allows teachers to powergrade student code, potentially affecting several times more students than would be possible without the tool.

Both of these tools allow staff members to have greater coverage of student code. But so do other tools that cluster student code. What separates these tools from those automated tools is that Comment Search and Code Search take advantage of human intelligence to nearly eliminate mistakes. Because this is a classroom setting, mistakes are fatal: a student who receives an incorrect comment may not realize that it is a mistake, and may learn the wrong thing. Comment Search and Code Search require that any comment given to a student is verified by an instructor as appropriate. While neither Comment Search nor Code Search give teachers the same relationships with students in an online classroom as with a traditional classroom, they take large steps towards improving feedback for students.

References

[1] Mason Tang. "Caesar: A Social Code Review Tool for Programming Education." M.Eng. Thesis, Massachusetts Institute of Technology, 2011.

[2] Basu, S., Jacobs, C., and Vanderwende, L. Powergrading: a Clustering Approach to Amplify Human Effort for Short Answer Grading. TACL 1, (2013), 391–402.

[3] Michael Brooks, Sumit Basu, Charles Jacobs, and Lucy Vanderwende. 2014. Divide and correct: using clusters to grade short answers at scale. In *Proceedings of the first ACM conference on Learning @ scale conference* (L@S '14). ACM, New York, NY, USA, 89-98.

[4] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," In Proc. of IWSC'09, 2009.

[5] John DeNero and Stephen Martinis. 2014. Teaching composition quality at scale: human judgment in the age of autograders. In *Proceedings of the 45th ACM technical symposium on Computer science education* (SIGCSE '14). ACM, New York, NY, USA, 421-426.

[6] Stephanie Rogers, Steven Tang, and John Canny. 2014. ACCE: automatic coding composition evaluator. In *Proceedings of the first ACM conference on Learning @ scale* (L@S '14). ACM, New York, NY, USA, 191-192.

[7] J. Huang, C. Piech, A. Nguyen, and L. Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In AIED 2013 Workshops Proceedings Volume, page 25, 2013.

[8] Karger, D. R., Oh, S. & Shah, D. (2011). Iterative Learning for Reliable Crowdsourcing Systems. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira & K. Q. Weinberger (eds.), *NIPS* (p./pp. 1953-1961).

[9] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (SIGMOD '03). ACM, New York, NY, USA, 76-85.

[10] Gail C. Murphy and David Notkin. 1996. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.* 5, 3 (July 1996), 262-292.

[11] Greg J. Badros. 2000. JavaML: a markup language for Java source code. *Comput. Netw.* 33, 1-6 (June 2000), 159-177.

[12] Robert C. Miller. *Lightweight Structure in Text.* PhD thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, May 2002. Published as CMU Computer Science technical report CMU-CS-02-134 and CMU

Human-Computer Interaction Institute technical report CMU-HCII-02-103.

[13] Roger F. Crew. 1997. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997* (DSL'97). USENIX Association, Berkeley, CA, USA, 18-18.

[14] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. 1996. Fast, Flexible Syntactic Pattern Matching and Processing. In *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)* (WPC '96). IEEE Computer Society, Washington, DC, USA, 144-.

[16] Terence Parr, The Definitive ANTLR 4 Reference, Pragmatic Bookshelf, 2013.