Caesar: A Social Code Review Tool for Programming Education

by

Mason Tang

S.B., Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 2011

Author ....
Department of Electrical Engineering and Computer Science
August 22, 2011

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

Accepted by ....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Caesar: A Social Code Review Tool for Programming Education

by

Mason Tang

Submitted to the
Department of Electrical Engineering and Computer Science

August 22, 2011

In partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Caesar is a distributed, social code review tool designed for the specific constraints and goals of a programming course. Caesar is capable of scaling to a large and diverse reviewer population, provides automated tools for increasing reviewer efficiency, and implements a social web interface for reviewing that encourages discussion and participation. Our system is implemented in three loosely-coupled components: a language-specific code preprocessor that partitions code into small pieces, filters out uninteresting ones, runs static analysis, and detects clusters of similar code; an incremental task router that dynamically assigns reviewers to tasks; and a language-agnostic web interface for reviewing code. Our evaluation using actual student code and a user study indicate that Caesar provides a significant improvement over existing code review workflows and interfaces. We also believe that this work contributes a modular framework for code reviewing systems that can be easily extended and improved.

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

Acknowledgements

I would like to thank my advisor, Rob Miller, whose guidance has been instrumental in keeping me on track and on task throughout the entire course of this thesis. He has been patient, responsive, and consistently receptive to even the most outlandish ideas to come out of our numerous brainstorming sessions. Most valuably to me, he has taught me a great deal about research, software interfaces, software engineering, and myself.

While Prof. Miller who gave me the guidance and direction for my thesis, I would not have made it very far without the constant and unwavering support of my family. My parents and my brother have always been a warm and loving home to me, whether that means a home-cooked meal, a friendly phone call, or even a short e-mail to see how my week has been. I am truly very lucky to have them, and I constantly look to their examples in an effort to become a better person.

Equally supportive throughout this project has been my girlfriend, Elena Kon, who, despite living across the country and three timezones away, has consistently managed to be one of my greatest sources of comfort and strength. She has been endlessly patient with me even at my moments of greatest stress, and has believed in me and my abilities even when I had managed to thoroughly convince myself otherwise.

I would also like to thank my roommates and best friends, Greg and Katrina, for giving me advice, calming me down when I needed it, and putting up with some extra dirty dishes when deadlines were looming. They have been, in almost every respect, a second family to me.

This thesis would not have made it very far either without the inspiration, encouragement, and example of the rest of the UID group. It has been my pleasure and honor to find such a group of exceptionally talented researchers to call my colleagues and friends. A special thanks as well to Elena Tatarchenko for joining the project and helping me with the implementation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Software code review is the seemingly simple practice of human examination of source code to improve software quality. This can range from informal “over the shoulder” code reviews to mandatory company-wide review policies supported by specialized code review tools. Regardless of its implementation, code review provides not only the direct effect of catching potential programmer errors, but also the indirect, and often more valuable, effect of spreading developer knowledge and expertise among those involved. Code review as an existing practice in the software engineering industry and open source community generally assumes: that the unit of code being reviewed is a set of changes against an existing codebase (incremental code review), that reviewer assignment is an ad hoc decision made by the submitter, and that the reviewers involved are also themselves developers familiar with the relevant code.

Meanwhile, code review as it exists in industry has found little traction in programming education. While other software engineering best practices from industry like the use of static analysis tools, version control systems, unit testing, pair programming, and rapid iterations have all experienced at least some adoption in the classroom, peer review of student source code has remained relatively unexplored. Much of this can be attributed to some of the additional constraints encountered in a classroom setting that challenge or violate the assumptions inherited from the standard code review model, such as: the possibility of plagiarism, the relative inexperience of students, and the task (or burden) of evaluating student performance. These constraints preclude the possibility of using an existing tool from industry for classroom code review, and indicate the need for a more specialized approach.

While existing review systems are built for small teams of experienced developers whose primary goal is to improve code quality, our problem is fundamentally different. A classroom code review tool needs to be built for a large group of students of varying levels together with a small group of experienced teaching staff. Furthermore, while code quality is important, the primary goal in a classroom is to teach the students how to be better software engineers. By designing a few key features into the system with the characteristics of our reviewer population in mind, we can accelerate and improve both the quality of the code review and the quality of the student learning that result. This thesis presents Caesar, a distributed, social code review system developed for programming education. Caesar allows for distributed review of student code submissions without requiring version control, is designed to take advantage of a large and diverse reviewer population, and provides teaching staff with the ability to monitor and moderate the review process.

For the purposes of our system, we consider each one of our users as belonging to one of three main roles:

Students

Students within our system are the authors of the code being reviewed, and also ultimately the recipients of all of the feedback generated in the code review process. They also function as reviewers themselves. Because the code review is happening within a classroom, the presence of students in our system also adds the dual constraints of maximizing opportunities for learning and minimizing the opportunities for abuse (plagiarism, laziness, etc.).

Staff

The teaching staff for the class are nominally responsible for their share of code review, but their primary function is to monitor the review process and provide instruction to students who need it. They are also expected to synthesize the feedback written and received by students as a part of the grading and evaluation process within the class.

Other

The rest of the users in the system are people recruited or invited to help review code. They can be alumni looking for a way to contribute back to MIT, alumni in industry looking for a way to connect to and recruit talented students, current students who took 6.005 in the past, or potentially even hired tutors.

Caesar is implemented in three decoupled components: a language-aware preprocessing tool for partitioning student submissions, generating automated feedback, and clustering similar code, a task routing component that automatically assigns chunks to reviewers, and a language-agnostic web application that users interact with. While we have only implemented a Java preprocessor in this iteration, Caesar's architecture allows for support for other languages with minimal effort.

To describe our system, we will use the following vocabulary:

Assignment

A single classroom assignment, such as a problem set or a project. There will typically be a handful of these per course per semester.

Submission

A student's solution for an assignment. This is what each student turns in for a given assignment. There will typically be one submission for each student for each assignment. A student solution consists of one or more files of source code.

Chunk

A small piece of code partitioned from a submission. This is the smallest unit of review in our system. In our current implementation, a chunk is typically a single method.

Task

A single unit of work, which in our system is equivalent to assigning one chunk to one reviewer.

After a programming assignment is due, the Caesar preprocessor automatically partitions student submissions into small chunks. Dividing student submissions into smaller pieces is crucial in giving the system the flexibility to distribute the reviewing load across a large population and to maximize reviewer efficiency by trying to assign the optimal chunks to each reviewer. We have also implemented a filtering system to try to remove chunks that are determined to be either trivial (stubs or empty functions) or redundant (staff-provided template code).

The preprocessor, after partitioning and filtering the code into a set of chunks, is also responsible for annotating them with feedback generated by static analysis tools. By incorporating feedback from static analysis inline in the review process, we can expose, for human review, problems that might be ignored, and we can increase reviewer efficiency by automatically generating comments for easily-identified problems that otherwise would require a reviewer to manually write a comment. The current implementation only supports the Checkstyle Java code style checker, but adding support for other tools is planned. The feedback generated from this step is displayed using largely the same interface as comments written by human reviewers.

The preprocessor then uploads the chunks into the web application, where they are stored and then dynamically routed to users for review. This step automates the reviewer assignment process and allows the system to scale and adapt to large, diverse reviewer populations. For our purposes the process of assigning users to review tasks, or *task routing*, is designed to maximize student-to-staff interaction and interaction between users of different abilities and roles, while simultaneously maintaining an even workload throughout reviewers of the same role and ensuring fairness in the amount of attention received by each student submission. Our task routing implementation accomplishes this based on several heuristics, and operates dynamically as users enter the system, since our user population is at least partially undetermined when an assignment is submitted for review.

The interface that our users use to review code has been designed to be as efficient and effective as possible in a classroom setting. At a basic level, the core reviewing interaction as implemented in our application is designed to be streamlined and usable, and on its own is a usability improvement on existing code review systems. In addition, recognizing that peer review is an inherently social task, we have built a number of features into the web interface that enable and incentivize interaction between users: threaded comments, a voting system, activity streams, and a reputation system. These features encourage users to participate and allow students to learn not only directly from the feedback they are given, but from interactions with their peers and with staff.

We evaluated our system using four assignments drawn from previous iterations of 6.813 User Interface Design and Implementation and 6.005 Elements of Software Construction. Results from the code preprocessor and the task routing component indicated that our system can effectively partition and distribute reviewing tasks for assignments with a high proportion of common template code, which allows the filtering and clustering code to produce better data to drive the task routing. Our system struggled with larger project assignments that allowed students more freedom in structuring their code, a problem that we leave to future work to address. A user study of our reviewing interface yielded mostly positive feedback on its usability, but indicated that adding more context to the code being reviewed could increase review quality.

In building Caesar, our main contributions are developing:

- A code review system specifically for classroom use
- A distributed, crowd-sourced code reviewing workflow
- A streamlined, social web interface for reviewing code

The remainder of this thesis covers the design and implementation of Caesar. Chapter 2 summarizes related work in the field. Chapters 3, 4, and 5 cover, respectively: our code preprocessor, our task routing system for mapping chunks to reviewers, and our end-user reviewing interface. Chapter 6 describes the evaluation of the preprocessor, the user interface, and the entire end-to-end system. Chapter 7 is the conclusion, and describes future work.

Chapter 2

Related Work

2.1 Code Review in Industry

Peer code review, where source code is manually read and inspected by other developers, is one of many processes developed originally in the software engineering industry to reduce the number of defects introduced into software. Originally a formal process with in-person meetings and complex procedures [10], many organizations now have implemented more lightweight processes that forgo face-to-face interactions, without significant reduction in defect reduction effectiveness [5]. Many development teams also use tools developed specifically to support peer code review, which can range from proprietary tools like Google's Mondrian [22] or Atlassian Crucible [8], to open-source efforts like Rietveld [25], Gerrit [12], and Review Board [24].

A common theme echoed by those who have studied the benefits of code review is the idea that peer review of code yields benefits beyond the direct effect of detecting and removing defects. By having a developer examine another developer's code, peer review can help spread domain expertise and improve code maintainability by forcing more developers to become familiar with code. The conversation surrounding the review process can help create a valuable shared understanding [34], something Fagan alluded to as product knowledge [10], among those involved.

Within industry, Chu-Carroll [3] acknowledges that code review benefits the development process by spreading knowledge, but goes on to argue that the "biggest advantage of code review is purely social." He states that, in an environment where code review is mandatory, the expectation that any code written will be examined by another pair of eyes is a powerful motivator to write better and more understandable code.

2.2 Code Review in Education

Despite being widely regarded in industry as a useful development process, peer code review has not gained significant adoption in the classroom [11, 20, 31, 33]. While a variety of high-quality solutions, like those mentioned earlier, exist in industry, there is a mismatch between the fundamental design choices of a traditional code review system and the constraints and requirements of a *pedagogical* code review system. Most existing tools in industry assume that the basic unit of review is a commit or patch made against a shared codebase, and none of them provide substantial support for automatically assigning reviewers. They generally support a workflow where: a developer submits a commit for review, the developer chooses one or two reviewers to review the code, the reviewers review the code, and after zero or more iterations, the code is submitted into the repository. These design assumptions limit the adoption of existing code review tools in the programming classroom due to the possibility of plagiarism, the necessity for fairness, and the difficulty of scaling reviewer assignment to a large class of students.

Motivated by these problems and the potential benefit of introducing code review in an educational context, there have been several notable projects and studies aimed at understanding and developing an effective code review system for classroom use. Several studies of peer review in programming classes and other fields of education have indicated that peer review can:

- Be closely correlated with expert graders in student evaluation [11, 19, 23]
- Help develop reviewing skills for the workplace [30]
- Improve the quality and understanding of student code over time [15, 23]

There have been similar tools developed specifically to support pedagogical code review, but we believe that the system described in this thesis improves on existing systems in the several key areas.

First, our reviewing interface represents a significant usability improvement over existing pedagogical code review systems. Some systems, like the work of Reily et al. described in [23] and RRAS by Trivedi et al. described in [31], are limited to structured feedback from students based on rubrics or forms designed by teaching staff. This approach, while it has its advantages, places additional overhead on the staff and removes the opportunity of preparing students to review code in industry where the preferred form of feedback is annotations on source code. Other systems that do support code annotations, like OSBLE by Hundhausen et al. [16], omit features like multi-line comments and in general are relatively difficult to use by modern web design standards.

Second, the combination of code partitioning and task routing in our system allows Caesar to scale much more effectively to a large number of reviewers of varying abilities than existing systems. In systems lacking an automated mechanism for assigning reviewers to student submissions [16, 31, 32], performing the same task manually can place a serious burden on the teaching staff, especially if non-student reviewers are allowed to participate. Existing systems that do include automated reviewer assignment [11, 23] do not implement any automated code partitioning strategy, which forces reviewers to examine an entire student submission. This can lead to wasted reviewer time, especially with larger assignments. We argue in this thesis that assigning reviewers to chunks instead of submissions gives us the flexibility to better optimize review efficiency and quality.

Third, our system is a *social* code review system, with features like discussion support, reputation, and activity streams, that is designed to encourage the inherently social nature of peer review. Other social systems have been successfully built for programmers, such as Stack Overflow [21], that leverage the idea of community and reputation. Existing code review systems, both in industry and in the classroom, only indirectly capitalize on the benefits of incentivizing participation and building community.

Finally, to our knowledge, there is no existing system that integrates automated commenting into the review process, and also no existing system that uses syntactic similarity of student code to guide the review process.

2.3 Crowdsourced Workflows

Crowdsourcing, a term first coined by Jeff Howe [14], describes the act of distributing a large set of small tasks to a large group of people. Caesar, as a system that allows for a crowd of students, teaching staff, and others to collectively review a body of student code, is a form of crowdsourced code review. To our knowledge, this is the first system that explicitly applies crowdsourcing techniques to peer code review.

Outside of software development, crowdsourced review was implemented in Bernstein et al.'s project, Soylent [1], a Microsoft Word plugin built on Amazon's Mechanical Turk API and TurKit [18]. Soylent enables end-users to quickly harness a crowd of human workers to proofread paragraphs and correct for mistakes. To improve result quality, Soylent implements a crowdsourced workflow pattern that its authors call *find-fix-verify*, a technique that structures the type and allocation of the assigned tasks to account for factors like lazy or unreliable workers. This idea of fundamentally structuring a large crowdsourced task to increase or ensure result quality in the face of a large crowd of imperfect workers is one that this thesis draws heavily on.

In large open-source development projects, a form of implicit ad hoc crowdsourcing emerges, where incoming reviews are broadcast to hundreds of potential reviewers, typically via e-mail. While this sounds chaotic, a study of existing projects has indicated that communities like this develop effective procedures for managing their code reviews. [26] In this approach, the unit of review is a patch or commit and the task routing strategy is entirely manual (either the contributor selects a reviewer for the code, or the reviewer picks up the review task from a broadcast). Ideally, a classroom tool would be more structured, as required by the necessity for student evaluation and an even workload distribution.

Cosley et al. [6] describe a system called SuggestBot designed to route Wikipedia users to articles in need to attention using a workflow pattern called *intelligent task routing*. They demonstrate that by intelligently matching users to the tasks that they are most likely to complete, they can increase the overall amount of useful work that users voluntarily contribute. Intelligent task routing has also been applied to peer review in the classroom before [7], but without a defined notion of what reviewer-author pairings are desirable. Our system applies the idea of task routing to code review, made possible through code partitioning, and strives to assign reviewers to student code to maximize goals like student learning and an even workload distribution.

2.4 Static Code Analysis

While largely outside of the main scope of this thesis, code analysis tools and techniques play an important role in Caesar's architecture. We rely on Checkstyle [2] to generate automated code style comments, and have implemented a version of the robust winnowing algorithm for document fingerprinting developed by Schleimer et al. [29] for the purposes of measuring code similarity.

Chapter 3

Code Preprocessing

When code is first submitted into Caesar, it passes through the preprocessor before being uploaded into the task router and ultimately the reviewing interface. The preprocessor is responsible for partitioning the code into small chunks, filtering out trivial or redundant chunks, running static analysis to generate automated comments, and detecting clusters of similar chunks. This component is implemented separately from the reviewing interface because several of its tasks, such as partitioning and clustering, are computationally intensive and more easily implemented as batch processes, and also to decouple language-specific code from the language-agnostic web application.

3.1 Partitioning

3.1.1 Design

The partitioner is the first stage of the preprocessor, and is responsible for turning potentially large student submissions into more manageably sized chunks. Manipulating these small pieces of code instead of entire student submissions at once gives us the flexibility to scale the code review process from the traditional model of including only a few reviewers at once to an entire class of over 100 students. It is important, however, to ensure that these gains in scalability and flexibility do not come at the cost of review quality; the chunks themselves must still be semantically meaningful to reviewers so that they have enough information to write useful feedback.

Because it is generally unfeasible for a single developer to review an entire project (or other functional unit) at a time regardless of setting, the idea of partitioning code into smaller units is an important component in any code review process. In a traditional model with a small team of professional developers working with version control, the unit of review is a single commit, or a set of file deltas. This type of *temporal partitioning* relies on the developers themselves organizing their code changes into reasonably sized deltas to the existing codebase. The commit is usually then reviewed by one or two other developers before being integrated.

Developing a code review model for programming education makes this type of code review impractical at best for two reasons: students, especially in an introductory class, are inexperienced with version control usage, and inviting other students to review code changes before an assignment is due makes plagiarism a real possibility. Instead, we are limited to partitioning the code after an assignment's deadline has passed, with student submissions being the input unit. We disregard any historical information contained in the version history of the submitted files, since we assume that, without good version control habits, it is difficult to meaningfully characterize the commits that students create. Therefore, we focus on *content partitioning*, where we split the code into semantically meaningful chunks based on the contents of the files only.

The design of the content partitioning in our system is critical in shaping the reviewing experience and the quality of the resulting feedback. An ideal partitioning, in a reviewing context, should be:

Understandable

Reviewers, when presented with a chunk, should be able to easily understand what it is without specialized background knowledge.

Complete

The information in the chunk should be sufficient to thoroughly evaluate the entire contents of the chunk.

Compact

The chunk should be as small as possible, to save reviewer time and increase the flexibility of the task routing. This means that we prefer many small chunks to fewer large chunks.

For this version of the code partitioning module for Java, we chose a relatively simple and well-understood method for determining what comprises a semantically meaningful chunk of code: individual methods from each class. This means that for a given Java class, our partitioning code will usually split it into one chunk for each of the constructors, and one chunk for each of the remaining methods.

This simplistic partitioning strategy is easily understood by reviewers, but suffers in terms of completeness and compactness. In many cases, a method calls other non-trivial and non-obvious pieces of code which make it difficult to review in isolation. The problem of summarizing that information along with other forms of surrounding context without sacrificing understandability and compactness is an open problem, and there are many potential solutions that could be implemented (see subsection 7.1.1). Partitioning by method also makes high-level design issues difficult or impossible to diagnose without reviewing many related chunks, an issue of compactness and completeness that could be remedied with more sophisticated code summarization techniques. Alternative partitioning schemes could present reviewers with information like class hierarchy trees, dependency graphs, or class files with method bodies removed (just fields and method prototypes).

3.1.2 Implementation

The actual partitioning task is handled by first crawling the input directory for all Java files, and then parsing all input Java code with the parser provided with the Eclipse Java Development Tools Core Component, extracted from an existing Eclipse installation. The resulting abstract syntax trees are then traversed with a visitor that constructs chunk objects when it encounters a constructor or method.

3.2 Chunk Filtering

3.2.1 Design

Ideally, with enough time, we would be able to review every chunk produced from the partitioning step. Realistically, we are highly constrained by the number of available reviewers and the amount of time they are willing to devote, and student submissions can often contain a large amount of code that is identical across submissions due to common staff template code being distributed with assignments.

3.2.2 Implementation

Our system implements a modular and extensible filtering mechanism that allows for chunks to be removed from consideration after the partitioning step. There are currently two types of filters in use: a chunk size filter, and a duplicate chunk filter. The chunk size filter discards chunks whose sizes fall below a certain threshold (5 lines, including whitespace and comments) to avoid wasting reviewer time on trivial pieces of code like getters, setters, and empty functions. Next, the duplicate chunk filter removes chunks that appear as exact duplicates (ignoring whitespace and letter case) more than some constant number of times (5) throughout the entire set of submissions, based on comparing the MD5 hash of the contents of the chunk with whitespace removed and ignoring letter case. We assume that these type of duplicate chunks are staff-provided template code, third-party libraries used by students, or (unfortunately) plagiarized solutions, and therefore do not need to be reviewed. This allows Caesar to automatically infer and remove common template code from student submissions without requiring staff to manually input the original template.

3.3 Automated Commenting

3.3.1 Design

After the code partitioning step, the filtered set of chunks is then analyzed to generate automated comments. One of the approaches our system takes in trying to minimize wasted reviewer time and maximize overall efficiency is to include automatically generated comments from static analysis tools in the review process so that human reviewers can save the effort of typing them (and perhaps just upvote or downvote them instead).

While static analysis tools are often deployed as build tools or IDE plugins in the development process, it can be difficult to standardize the development environment for a larger programming class (over 100 students) compared to a small team of a handful of developers without substantial effort from the staff. More serious is the reality that many developers, even if they choose to use static analysis tools, simply ignore their output if there are too many false positives or if they are pressed for time. Shifting these comments into the review process, as we have done with our automated commenting workflow, allows us to utilize a crowd of human reviewers to filter false positives and reframes the detected issues in a learning context.

3.3.2 Implementation

For the Java language alone, there several widely used and deployed analysis tools designed to reduce programmer error and improve code quality. In our current implementation, we have only included comments generated by the Checkstyle tool, which enforces coding standards and style guides. Our Checkstyle configuration is identical to the default set of rules based on the Sun code conventions [4] shipped with version 3.4 of the tool, except with all whitespace-related rules disabled. Listing 3.1 and Table 3.1 together show a small piece of example code and the corresponding Checkstyle violations. We hope to include other tools in the future as time allows, such as FindBugs, unit test results, code coverage analysis, and others. In a traditional code review tool, reviewers often waste their time making mechanical and superficial comments about code style issues like incorrect whitespace usage, bracing style, and identifier naming conventions. With Caesar, many of these issues are identified and presented to the reviewer automatically, avoiding tedious commenting and allowing them to simply express dissent or consent through our voting mechanism.

```
1 public static int mod_impl(int x, int y)
2 {
3     int result = x%y;
4     if (result < 0)
5         result += y;
6     return result;
7 }
```

Listing 3.1: Example Checkstyle input

Line	Comment
1	Parameter `x` should be final
1	Parameter `y` should be final
1	Name ‘mod_impl’ must match pattern ‘`^[a-z][a-zA-Z0-9]*$`’
1	Missing a Javadoc comment
2	‘`{`’ should be on the previous line
5	‘`if`’ construct must use ‘`{}`’s

Table 3.1: Checkstyle violations for example code in Listing 3.1

3.4 Clustering

3.4.1 Design

In an effort to increase reviewer efficiency, Caesar has the capability to identify clusters of similar code within an assignment. This information is later fed into the task routing code and used to try to assign similar chunks of code to the same reviewer. In practice, this typically results in a reviewer being assigned multiple student implementations for the same method specification. This design decision is motivated by the observation from existing grading practices that graders are often much more efficient when grading many solutions to the same problem, which minimizes mental context switching, leading to the common practice of splitting a grading task by problem rather than submission. Our clustering feature aims to achieve the same efficiency boost in a distributed chunk context.

3.4.2 Implementation

Before clustering the chunks, we generate a set of fingerprints for each with our implementation of robust winnowing [29]. A fingerprint for a chunk is an MD5 hash of an *n*-gram that occurs in the chunk contents. The robust winnowing algorithm searches the set of potential fingerprints for a chunk and selects a small subset of those fingerprints in a consistent way to preserve recall. For our implementation, we use *n*-grams of size 10 and a window of size 20 for winnowing. We then cluster the chunks using a freely available implementation [13] of the hierarchical agglomerative clustering algorithm.

In order to cluster the chunks using an existing clustering algorithm, however, we must first define a distance metric between any two chunks. Our implementation uses a combination of the Jaccard distance, $J_\delta(c_1, c_2) = 1 - \frac{|F(c_1) \cap F(c_2)|}{|F(c_1) \cup F(c_2)|}$, computed over the fingerprints of each chunk, $F(c)$, and a constant penalty, k_{np}, if the method names of the chunks differ. The Jaccard distance in our distance metric is a measure of how many fingerprints are unique to either chunk, where 0 indicates that both chunks have the same fingerprints, and a score of 1 indicates that neither chunk has any fingerprints in common. The name penalty exists because we assume that two methods that share both a class name and method name must be at least somewhat similar, despite how much their bodies might differ.

Finally, our implementation uses the minimum pairwise distance between chunks as its distance measure between chunk clusters, an approach known as *single linkage* in the machine learning literature. Intuitively, this means that a chunk need only resemble one chunk in a cluster to be included in that cluster. We use single linkage in order to make our clustering strategy as forgiving as possible.

Chapter 4

Task Routing

4.1 Design Goals

Splitting student submissions into small chunks gives us the opportunity to produce fine-grained reviewer assignments that can potentially yield increased reviewer efficiency and higher quality reviews overall. Our system implements a *task routing* mechanism for automatically and dynamically allocating chunks to reviewers that considers both reviewer characteristics (role, reputation) and chunk characteristics (similarity clusters). For our purposes, we will refer to a single assignment of a reviewer to a chunk as one *task*.

The design of our task routing strategy is guided by three major goals for the resulting assignments:

Efficiency

The resulting task assignment should attempt to maximize the overall efficiency of the review process. This includes considerations like assigning similar chunks to the same reviewer.

Quality

We also seek to maximize the overall quality of the resulting feedback and discussion. Some of the heuristics we consider in the pursuit of quality include reviewer interaction diversity and staff distribution.

Fairness

We recognize the necessity to ensure fairness wherever possible in the process. This involves evenly distributing workload across students, evenly distributing reviewers (especially staff) across student submissions, ensuring that students have the same discussion and feedback opportunities presented to them, and preventing abuse and plagiarism.

The system is also constrained by several implementation requirements. Because Caesar is designed with a large crowd of reviewers where the size and composition of the reviewer population, especially considering alumni reviewers, is not known in advance, our task routing algorithm must be able to assign tasks dynamically and iteratively as reviewers enter the system. This prevents us from using approaches that might try to globally optimize the task assignment solution across all reviewers and chunks. Instead, our system uses an incremental greedy algorithm that assumes nothing about the reviewer population initially.

Because our reviewers enter the system through the web interface and we would like to avoid noticeable delays, our task routing implementation must present them with their task assignments in under 1 second, ideally even faster. This, along with the large number of chunks, limits us to computationally simple heuristic approaches that can be executed quickly, instead of more exhaustive search or optimization strategies. It also requires that we perform the majority of our work in memory, instead of relying on database queries.

4.2 Implementation

The task routing process is enabled after an assignment has been uploaded to the web interface. Because our algorithm is incremental, we assign tasks to users only after they visit the interface for the first time since the assignment was uploaded. Teaching staff are an exception to this, and are assigned tasks after all other users have entered the system in order to best distribute staff attention to students in light of all of the existing task assignments.

Based on our goals and implementation constraints, the Caesar task routing algorithm uses a series of computed heuristics that it attempts to optimize one by one. In practice, to produce a chunk assignment, this means that the list of potential chunk assignments is progressively sorted by each computed value and then the top-ranking chunk is assigned. Several of the scores used are dependent on existing assignments, so the values are recomputed and the list is resorted if more task assignments are required. A simplified version of the code for the assignment routine is shown in Listing 4.1. The sorting key used for

```
def assign_next_task(user, chunks):
    sort_key = make_chunk_sort_key(user)
    chunk_to_assign = min(chunks, key=sort_key)
    return chunk_to_assign
```

Listing 4.1: Simplified code for task routing

the chunk ordering step is the most important part of the behavior of the task routing code. For each chunk, we generate a tuple of values as the sort key, with each value representing a heuristic in descending priority. For all users with the exception of teaching staff, these are, for a chunk:

1. Code cluster score
2. Number of reviewers already assigned to the chunk
3. Number of reviewers already assigned to the chunk's submission
4. Total user affinity over the submission
5. Total user affinity over the chunk

First, the code cluster score is a value that prioritizes chunks that belong to the same similarity cluster (computed by the preprocessing step) as another chunk that the user has already been assigned. This forces the system to try to assign code from the same cluster to the user. To avoid situations where reviewers have to review a large amount of similar code and therefore lose interest in the review process, we limit the assignment to at most 3 chunks for each cluster for each user. In addition, we also prioritize chunks from larger clusters, with the reasoning that those large clusters of similar code are the common required portions of the assignment, and therefore most important to review. We prioritize code similarity before all other metrics with the reasoning that reviewer time and attention are extremely limited, and decreasing the amount of time required to review a chunk by capitalizing on parallel structure allows our system to effectively review more code.

Second, we look for chunks that have the most currently assigned reviewers, ignoring chunks that already meet or exceed the configured maximum number of reviewers allowed per chunk (in our deployment this is set to 2). This behavior effectively causes reviewers to be grouped together on chunks, encouraging discussion and maximizing the amount of user interaction.

Third, we then look for chunks belonging to submissions with the fewest distinct reviewers, which forces the task assignment to evenly distribute reviewers across student submissions.

Fourth, we compute the sum of a value called the *user affinity* between the assignee and each of the users already assigned to the chunk. The user affinity is a pairwise measure between two users of, broadly speaking, how valuable their interaction could potentially be, and is defined as a sum of three components: distance affinity, reputation affinity, and role affinity. Distance affinity is negative for users that have already been assigned to the same chunk together, and zero otherwise. This is designed to increase the number of distinct reviewers that the user has an opportunity to co-review chunks with. The second term, reputation affinity, is simply the absolute value of the difference in reputation scores of the two users, which acts to increase the diversity of users assigned to each chunk. The third term, role affinity, is defined to be moderately positive when one of the users is staff and the other is a student, strongly negative when both are staff, weakly positive when the roles differ but are not either of the previous two cases, or 0 if the roles are the same. This term is designed to maximize the amount of students who are assigned to the same chunk as a member of the teaching staff, and as another means to increase the diversity of reviewers on each chunk.

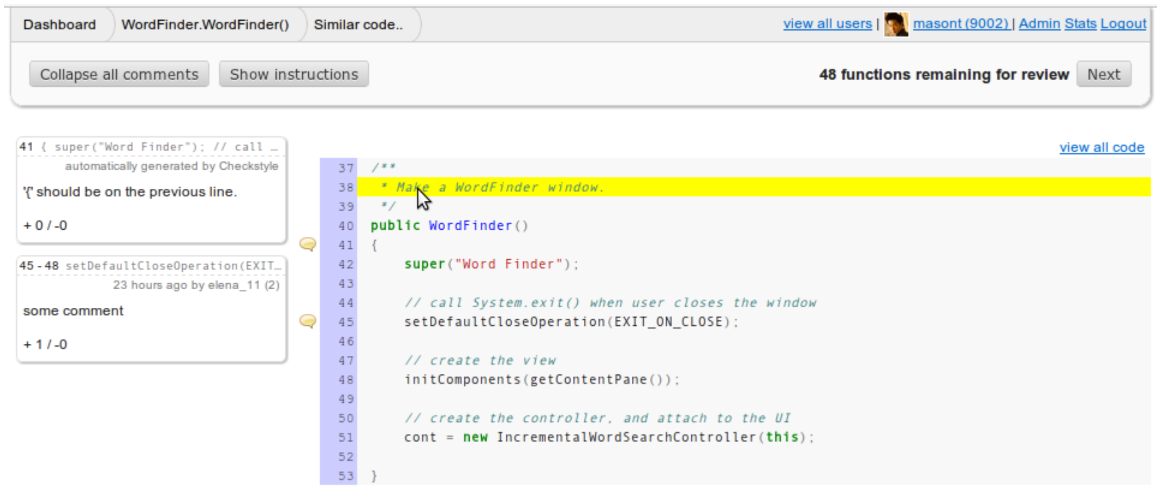
Fifth and finally, we compute the sum of the user affinity values between the assignee and the set of distinct reviewers assigned to any chunk belonging to the submission associated with the candidate chunk. Whereas previously we looked at the total affinity for just the candidate chunk itself, and attempted to maximize the diversity of interaction for the assignee, this step attempts to maximize the diversity of the users for the *submitter*. This step is crucial in ensuring that each student's submission gets a fair amount of staff attention, high reputation users, and simply total unique reviewers.

Because the distribution of staff throughout the system and their interactions with other users is more important than the actual reviewing experience of the staff themselves, when assigning tasks to them we ignore the first three steps outlined above and sort chunks only using the total user affinity of each chunk and then the total user affinity of its associated submission.

Chapter 5

Reviewing Interface

All end-user interactions within the Caesar system happen in the web interface, which is a Python web application built with the Django Framework, with substantial amounts of HTML, CSS, and Javascript code as well. The web application handles the fundamental tasks of allowing reviewers to review code, allowing students to access their feedback, and allowing staff to monitor and evaluate the process.



41 { super("Word Finder"); // call ...
automatically generated by Checkstyle
'{' should be on the previous line.
+ 0 / -0

45 - 48 setDefaultCloseOperation(EXIT...
23 hours ago by elena_11 (2)
some comment
+ 1 / -0

```
37 /**
38  * Make a WordFinder window.
39  */
40 public WordFinder()
41 {
42     super("Word Finder");
43
44     // call System.exit() when user closes the window
45     setDefaultCloseOperation(EXIT_ON_CLOSE);
46
47     // create the view
48     initComponents(getContentPane());
49
50     // create the controller, and attach to the UI
51     cont = new IncrementalWordSearchController(this);
52
53 }
```

Figure 5-1: Caesar's reviewing interface

5.1 Design Goals

The design of the reviewing application focuses on maximizing the amount of quality content in the system. We consider a piece of content to be of *high quality* if it satisfies the following four requirements:

Timely

Content should be submitted to the system as soon as possible, so that the review process can be efficient and quickly return feedback to the student. Timely comments and responses also help encourage real discussions between users.

Relevant

Feedback left for a region of code should be relevant and on topic. Replies to comments should be relevant to the parent comment.

Correct

The efficacy of our system would be vastly diminished if users could not trust the accuracy of the information being presented.

Informative

In order to reduce the amount of information clutter, we prefer comments that not only identify problems, but provide solutions to them and help students learn from their mistakes. This also means that we favor substantial comments with interesting information (e.g. "I would use LinkedList here instead of ArrayList based on your implementation strategy") instead of superficially correct comments (e.g. "You misspelled a word in this comment.").

The interface design tries to maximize these metrics with three main complementary design strategies:

Reduce overhead

When possible, contributing to the system should be as easy as possible. The interface should also attempt to minimize the amount of extraneous information and interface elements presented to the user.

Reward participation

Users should have a visible and definite incentive beyond goodwill to contribute good content.

Present relevant information

The interface should, as much as possible, attempt to identify and present relevant and high quality content to the user. This helps the user learn from that content, and feeds back into the system in the form of more informed contributions.

Because Caesar is designed for users belonging to one of three user roles (student, staff, other), we consider the possible primary use cases for all of these roles:

Reviewing code (students, staff, other)

The primary function of the tool is to enable users to leave feedback in the form of comments and votes on pieces of student code. All user roles share this use case.

Reading submission feedback (students, staff)

The students need an efficient way to read the feedback given to them on their own submissions. As a part of the grading process, the staff also need a way to see feedback given to students on a submission level.

Reading user contributions (staff)

In addition to the feedback given to a student, another part of the grading process is the feedback generated by a student. Staff need to be able to view all of the contributions generated by a user in a summarized view. Other users are only be able to see this information for themselves.

5.2 Features

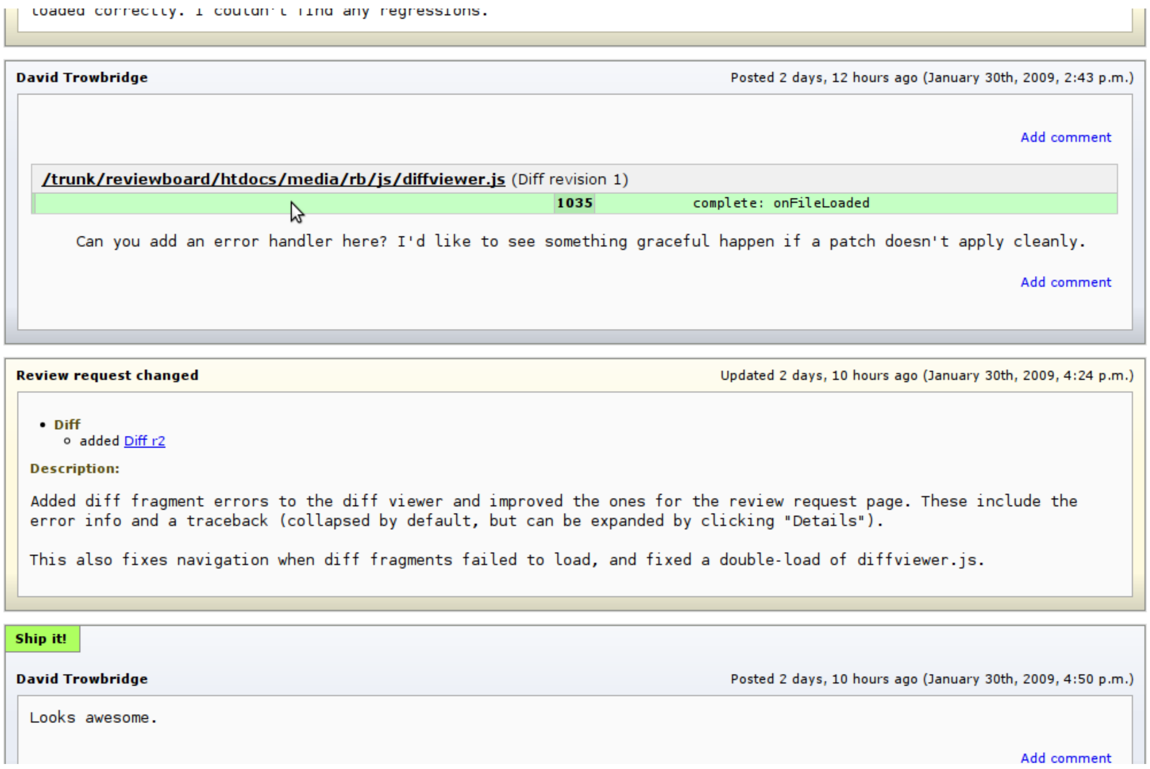
While the web interface is only one component of the entire system, it contains a number of unique features designed to facilitate the code review process. None of these are novel on their own, and in fact can be found on many other web applications, but few of them have been integrated into a code review system before, and none of them have been integrated into an existing classroom code review tool.

5.2.1 Streamlined Commenting Interface

The most important interaction that our interface is designed to support is leaving comments on pieces of code. Because of this, we have designed a commenting interface that is as streamlined and usable as possible. While the student and staff members of our user population can be somewhat easily motivated, by grades or other means, to participate and comment on code, the other members, who are essentially volunteers, are more difficult to encourage. Making the commenting workflow simple and painless is crucial in maximizing their participation.

Display

Unlike existing systems which typically only allow comments to be attached to a single line of code and display comments either separately (see Figure 5-2) or embedded in the flow of the source code (see Figure 5-3), our reviewing interface displays comments to the left of the source code in a style similar to document annotations as in Google Documents or Microsoft Word, as shown in Figure 5-1. In some cases, the vertical space occupied by the comments can be larger than the vertical space occupied by the code itself. This is especially true when comments generated by automated tools are included. This quickly causes comments to be visually separated from the regions of code they are annotating. The interface ameliorates this problem in two ways. First, the comments themselves are displayed with a small snippet of code context that can help the user contextualize the feedback without referring back to the chunk itself (Figure 5-5). Second, clicking a comment will cause the comment to scroll to align with its relevant lines of code. The interface also places comment markers alongside the source code indicating the presence of one or more comments (Figure 5-4). Clicking these markers will also scroll the relevant comments into view.



```
loaded correctly. I couldn't find any regressions.
```

David Trowbridge Posted 2 days, 12 hours ago (January 30th, 2009, 2:43 p.m.)

Add comment

/trunk/reviewboard/htdocs/media/rb/js/diffviewer.js (Diff revision 1)

```
1035            complete: onFileLoaded
```

```
Can you add an error handler here? I'd like to see something graceful happen if a patch doesn't apply cleanly.
```

Add comment

Review request changed Updated 2 days, 10 hours ago (January 30th, 2009, 4:24 p.m.)

- **Diff**
  - added Diff r2

Description:

```
Added diff fragment errors to the diff viewer and improved the ones for the review request page. These include the
error info and a traceback (collapsed by default, but can be expanded by clicking "Details").

This also fixes navigation when diff fragments failed to load, and fixed a double-load of diffviewer.js.
```

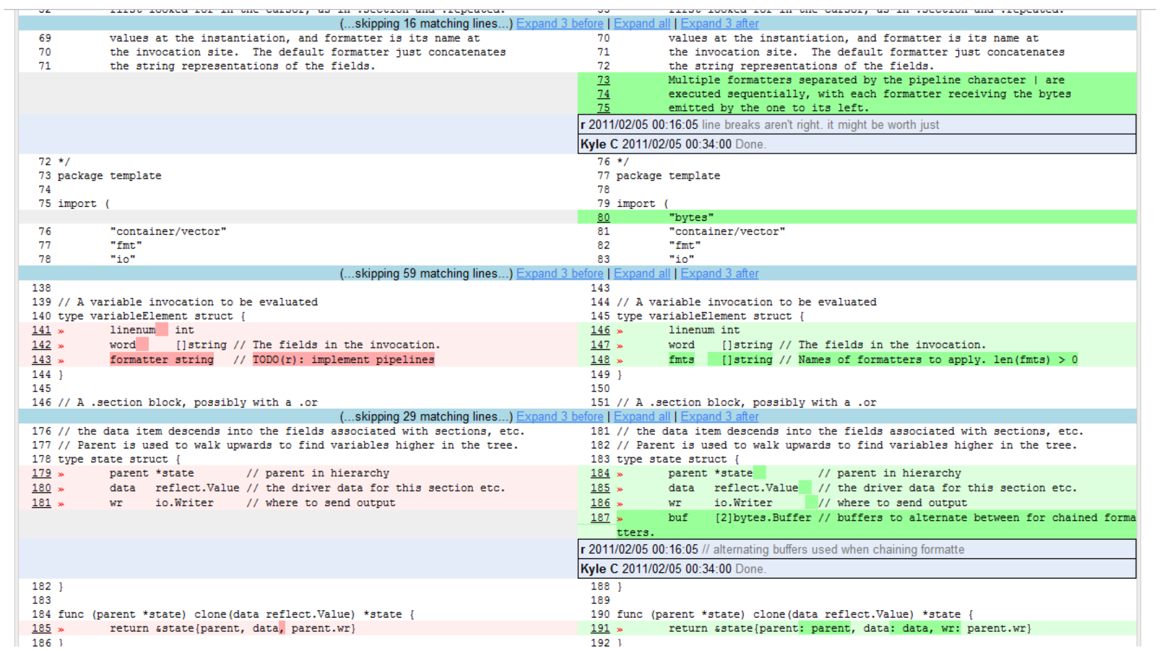
Ship it!

David Trowbridge Posted 2 days, 10 hours ago (January 30th, 2009, 4:50 p.m.)

```
Looks awesome.
```

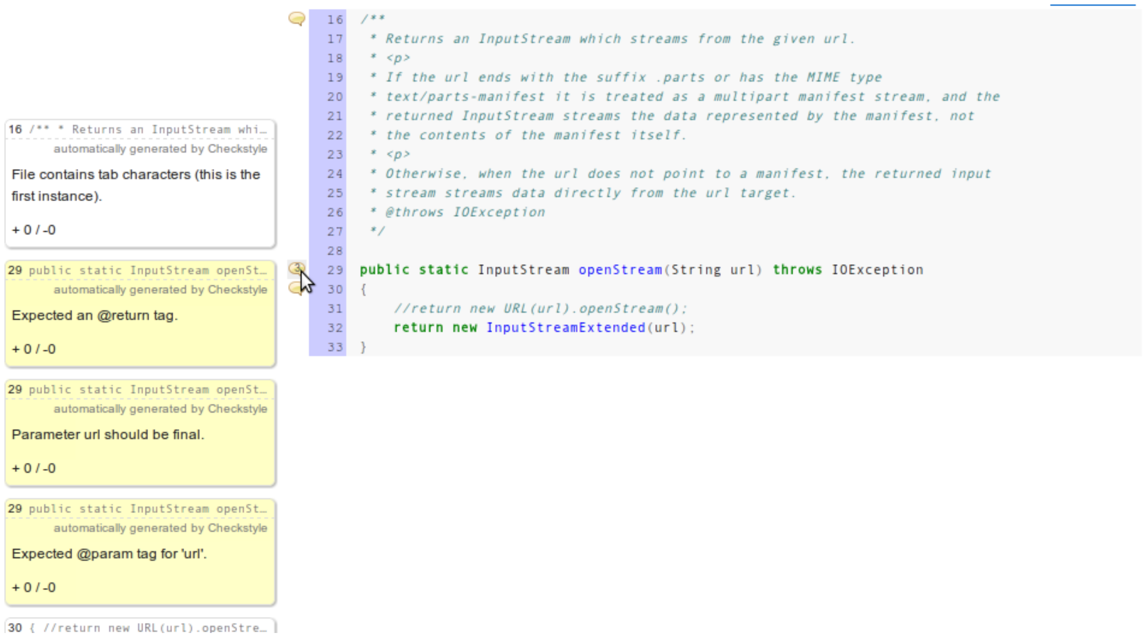
Add comment

Figure 5-2: Review Board comment discussion interface



```
(...skipping 16 matching lines...) Expand 3 before | Expand all | Expand 3 after
 69         values at the instantiation, and formatter is its name at         70         values at the instantiation, and formatter is its name at
 70         the invocation site.  The default formatter just concatenates     71         the invocation site.  The default formatter just concatenates
 71         the string representations of the fields.                         72         the string representations of the fields.
                                                                              73         Multiple formatters separated by the pipeline character | are
                                                                              74         executed sequentially, with each formatter receiving the bytes
                                                                              75         emitted by the one to its left.
                                                                              r 2011/02/05 00:16:05 line breaks aren't right. it might be worth just
                                                                              Kyle C 2011/02/05 00:34:00 Done.
 72 */                                                                        76 */
 73 package template                                                          77 package template
 74                                                                           78
 75 import (                                                                  79 import (
                                                                              80         "bytes"
 76         "container/vector"                                                81         "container/vector"
 77         "fmt"                                                             82         "fmt"
 78         "io"                                                              83         "io"
(...skipping 59 matching lines...) Expand 3 before | Expand all | Expand 3 after
138                                                                          143
139 // A variable invocation to be evaluated                                 144 // A variable invocation to be evaluated
140 type variableElement struct {                                            145 type variableElement struct {
141 »       linenum  int                                                     146 »       linenum int
142 »       word      []string // The fields in the invocation.              147 »       word    []string // The fields in the invocation.
143 »       formatter string   // TODO(r): implement pipelines               148 »       fmts    []string // Names of formatters to apply. len(fmts) > 0
144 }                                                                        149 }
145                                                                          150
146 // A .section block, possibly with a .or                                 151 // A .section block, possibly with a .or
(...skipping 29 matching lines...) Expand 3 before | Expand all | Expand 3 after
176 // the data item descends into the fields associated with sections, etc. 181 // the data item descends into the fields associated with sections, etc.
177 // Parent is used to walk upwards to find variables higher in the tree.  182 // Parent is used to walk upwards to find variables higher in the tree.
178 type state struct {                                                      183 type state struct {
179 »       parent *state        // parent in hierarchy                      184 »       parent *state           // parent in hierarchy
180 »       data   reflect.Value // the driver data for this section etc.    185 »       data   reflect.Value    // the driver data for this section etc.
181 »       wr     io.Writer     // where to send output                     186 »       wr     io.Writer        // where to send output
                                                                         187 »       buf    [2]bytes.Buffer // buffers to alternate between for chained forma
                                                                             tters.
                                                                         r 2011/02/05 00:16:05 // alternating buffers used when chaining formatte
                                                                         Kyle C 2011/02/05 00:34:00 Done.
182 }                                                                        188 }
183                                                                          189
184 func (parent *state) clone(data reflect.Value) *state {                  190 func (parent *state) clone(data reflect.Value) *state {
185 »       return &state{parent, data, parent.wr}                           191 »       return &state{parent: parent, data: data, wr: parent.wr}
186 }                                                                        192 }
```

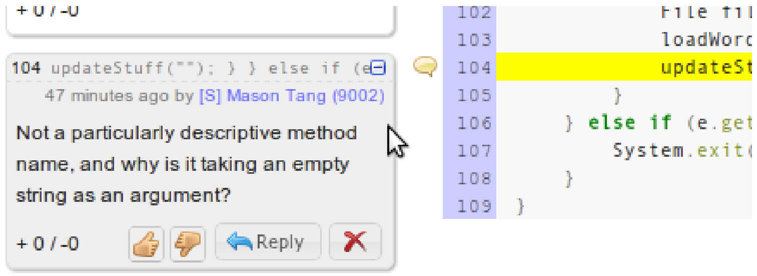
Figure 5-3: Rietveld review summary interface



```
16 /** * Returns an InputStream whi…
        automatically generated by Checkstyle
File contains tab characters (this is the
first instance).
+ 0 / -0

29 public static InputStream openSt…
        automatically generated by Checkstyle
Expected an @return tag.
+ 0 / -0

29 public static InputStream openSt…
        automatically generated by Checkstyle
Parameter url should be final.
+ 0 / -0

29 public static InputStream openSt…
        automatically generated by Checkstyle
Expected @param tag for 'url'.
+ 0 / -0

30 { //return new URL(url).openStre…
```

```java
/**
 * Returns an InputStream which streams from the given url.
 * <p>
 * If the url ends with the suffix .parts or has the MIME type
 * text/parts-manifest it is treated as a multipart manifest stream, and the
 * returned InputStream streams the data represented by the manifest, not
 * the contents of the manifest itself.
 * <p>
 * Otherwise, when the url does not point to a manifest, the returned input
 * stream streams data directly from the url target.
 * @throws IOException
 */

public static InputStream openStream(String url) throws IOException
{
    //return new URL(url).openStream();
    return new InputStreamExtended(url);
}
```

Figure 5-4: Comment markers on a chunk



```
+ 0 / -0

104 updateStuff(""); } } else if (e
    47 minutes ago by [S] Mason Tang (9002)
Not a particularly descriptive method
name, and why is it taking an empty
string as an argument?
+ 0 / -0
```

```
            File fil
            loadWord
            updateSt
        }
    } else if (e.get
        System.exit(
    }
}
```

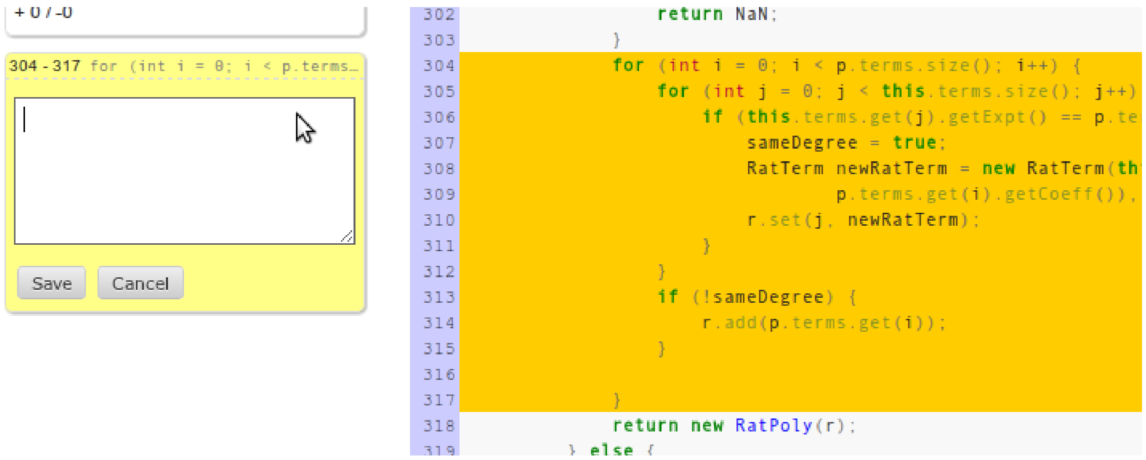
Figure 5-5: Comment display and discussion interface

Comment Entry

To leave a comment, users click and drag a region of code. When they release the mouse button, a comment form is displayed adjacent to the region and inserted into the list of comments on the left where the newly created comment would appear (Figure 5-6). The text field is given keyboard focus to minimize unnecessary user input.

5.2.2 Discussion Support

Beyond the basic use case of leaving a comment on a region of code, the web interface also supports several features designed to encourage communication between reviewers, in addition to communication between the reviewers and the submitter. First, users can



```
                return NaN;
            }
            for (int i = 0; i < p.terms.size(); i++) {
                for (int j = 0; j < this.terms.size(); j++)
                    if (this.terms.get(j).getExpt() == p.te
                        sameDegree = true;
                        RatTerm newRatTerm = new RatTerm(th
                                p.terms.get(i).getCoeff()),
                        r.set(j, newRatTerm);
                    }
                }
                if (!sameDegree) {
                    r.add(p.terms.get(i));
                }

            }
            return new RatPoly(r);
        } else {
```

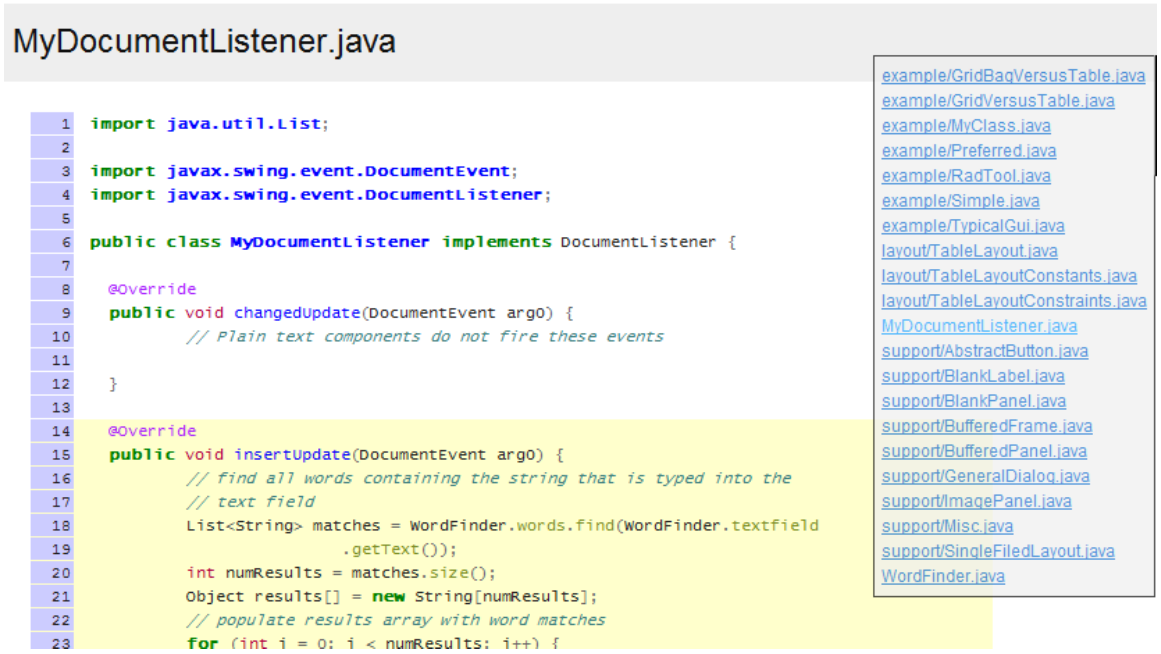
Figure 5-6: New comment form for writing a comment

reply to any existing comment, allowing for discussions and debates about contentious issues, question and answer, and other forms of interactions. Comment threads are limited to one level deep only, in order to preserve the limited horizontal space in the interface and to simplify the user experience.

Second, users can also either *upvote* or *downvote* existing comments in the system. This mechanism gives users a way to indicate both consensus and quality content. In the simple case where users find existing comments that they agree with, votes give a simple method to record that agreement, instead of forcing them to self-censor in an effort to reduce duplicated content. In other cases, votes can give other users an indication that a piece of feedback is particularly insightful or helpful. Finally, when coupled with a reputation system, a voting system acts as an incentive to encourage users to contribute relevant high quality content, and provides a way for users to quickly identify it as well.

5.2.3 Code Display

When viewing the source code for a chunk, reviewers are presented with a syntax-highlighted version of the code along with line numbers relative to the chunk's context in its original file. We also provide a read-only view for reviewers to examine the entire submission that a chunk was originally partitioned from. To avoid cluttering the interface, this view is hidden behind a link on the chunk reviewing interface. The submission code is organized by file with a collapsible file explorer to easily jump between files and with chunk boundaries clearly highlighted.



```java
import java.util.List;

import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;

public class MyDocumentListener implements DocumentListener {

  @Override
  public void changedUpdate(DocumentEvent arg0) {
          // Plain text components do not fire these events

  }

  @Override
  public void insertUpdate(DocumentEvent arg0) {
          // find all words containing the string that is typed into the
          // text field
          List<String> matches = WordFinder.words.find(WordFinder.textfield
                          .getText());
          int numResults = matches.size();
          Object results[] = new String[numResults];
          // populate results array with word matches
          for (int i = 0; i < numResults; i++) {
```

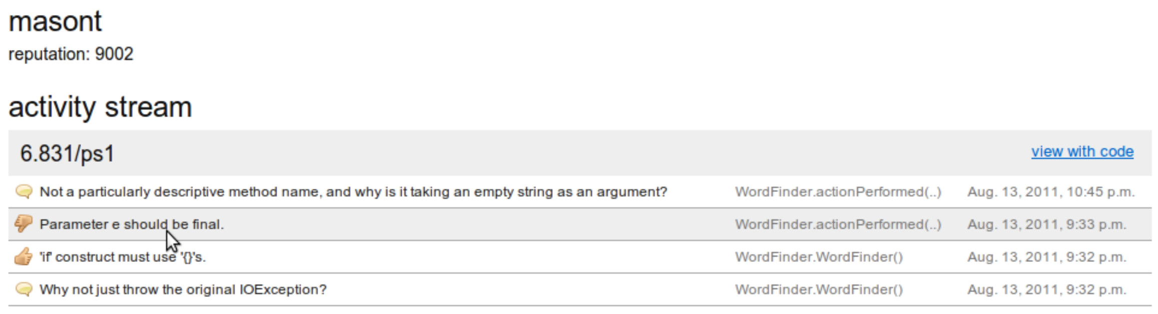
Figure 5-7: Interface for browsing submission code

5.2.4 Activity Streams

In order to facilitate ongoing interactions within the system (discussions on chunks, votes on comments) and to expose users to new and relevant content, Caesar can automatically find relevant content and present it to the user in reverse chronological order, a user interface pattern known as an *activity stream*. While other systems can sometimes have complex and opaque heuristics for determining the relevancy of stream content (Facebook, etc), Caesar provides an easily understandable set of rules for determining what content is relevant to the user. The dashboard activity stream as implemented displays, for the logged in user, the following actions:

- Activity (comments, replies, votes) on chunks authored by the user
- Activity (replies, votes) on comments authored by the user
- Activity (comments, replies, votes) on chunks assigned to the user for review

In addition to presenting relevant activity happening throughout the community to a user, we also use the activity stream interface pattern to display all of the activity generated by a single user. Every user can see a stream of all of their own contributions, including comments, replies, and votes, in reverse chronological order (Figure 5-8). Staff also have the ability to view user activity streams for other users in the system, for the purposes of moderating activity and also assessing the participation of students in the class.



masont

reputation: 9002

activity stream

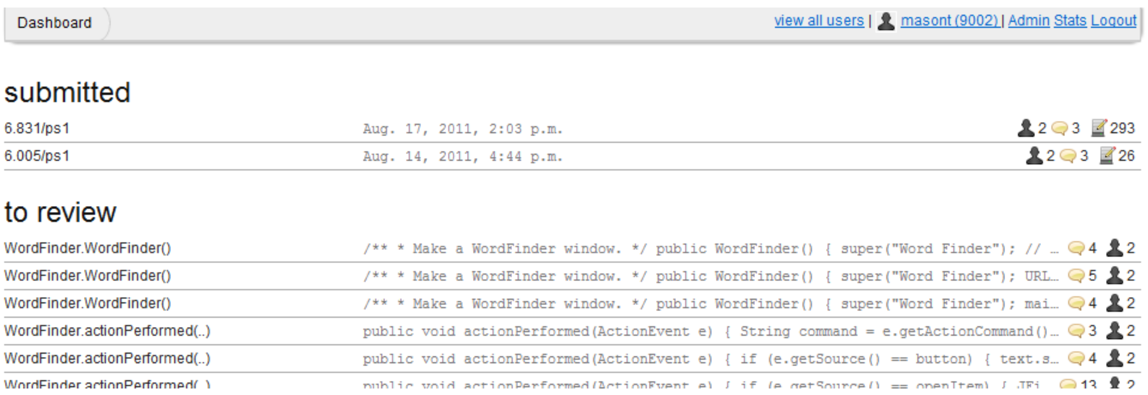
6.831/ps1 view with code

Not a particularly descriptive method name, and why is it taking an empty string as an argument?	WordFinder.actionPerformed(..)	Aug. 13, 2011, 10:45 p.m.
Parameter e should be final.	WordFinder.actionPerformed(..)	Aug. 13, 2011, 9:33 p.m.
'if' construct must use '{}'s.	WordFinder.WordFinder()	Aug. 13, 2011, 9:32 p.m.
Why not just throw the original IOException?	WordFinder.WordFinder()	Aug. 13, 2011, 9:32 p.m.

Figure 5-8: User contribution activity stream

5.2.5 Dashboard

When users first log in to the system, they are presented with a dashboard interface designed to surface and summarize the most relevant information in the system to them automatically. The dashboard displays submissions (for students) and task assignments. Throughout the reviewing process, we can safely expect students to be interested in the feedback they are receiving on their submissions, so the dashboard provides shortcuts to those. Since all users are responsible for at least some review tasks, the dashboard prominently displays them as an “inbox.” New or “unread” tasks that the user has not looked at yes are displayed as bold lines, using the familiar metaphor of an e-mail inbox that our users will almost certainly be familiar with.



Dashboard

view all users | masont (9002) | Admin Stats Logout

submitted

6.831/ps1	Aug. 17, 2011, 2:03 p.m.	2 3 293
6.005/ps1	Aug. 14, 2011, 4:44 p.m.	2 3 26

to review

WordFinder.WordFinder()	`/** * Make a WordFinder window. */ public WordFinder() { super("Word Finder"); // …`	4	2
WordFinder.WordFinder()	`/** * Make a WordFinder window. */ public WordFinder() { super("Word Finder"); URL…`	5	2
WordFinder.WordFinder()	`/** * Make a WordFinder window. */ public WordFinder() { super("Word Finder"); mai…`	4	2
WordFinder.actionPerformed(..)	`public void actionPerformed(ActionEvent e) { String command = e.getActionCommand()…`	3	2
WordFinder.actionPerformed(..)	`public void actionPerformed(ActionEvent e) { if (e.getSource() == button) { text.s…`	4	2
WordFinder.actionPerformed(..)	`public void actionPerformed(ActionEvent e) { if (e.getSource() == openItem) { JFi`	13	2

Figure 5-9: Dashboard interface with submissions and task assignments

Chapter 6

Evaluation

To evaluate Caesar using data that most accurately represents its intended use, we used 4 different assignments with student submissions taken from previous iterations of 6.813 User Interface Design and Implementation and 6.005 Elements of Software Construction:

WordFinder
: A 6.813 problem set where students were asked to implement a simple dictionary application. They were given some code to start with.

RatPoly
: A 6.005 problem set where students were asked to fill in parts of the implementation of a class for representing a rational polynomial.

Multipart
: A small 6.005 project where students, in groups of three, were asked to implement a multipart downloader. They were given some code to start with.

Antibattleship
: A large 6.005 project where students, in groups of three, were asked to implement a networked, graphical, multiplayer game called Antibattleship. They were not given any code to start with.

Table 6.1 shows some of the statistics relating to each of the four assignments. We note that we include comments and whitespace when counting lines of code. Most importantly,

			Per Submission	
Assignment	Students	Submissions	Files	Lines
WordFinder	70	70	3.0	889.9
RatPoly	114	114	14.2	3640.8
Multipart	120	40	14.3	1286.9
Antibattleship	99	33	75.7	9356.9

Table 6.1: Assignment statistics

we see that student submissions for Antibattleship were much larger than submissions for the other assignments, even when compared with the other group project, Multipart. In addition, we see that WordFinder submissions tended to be much smaller. Finally, we see that RatPoly had about three times as much code per submission as Multipart with the same number of files per submission, despite being an individual assignment.

6.1 Code Preprocessor

6.1.1 Partitioning

Before implementing Caesar, we ran two small preliminary studies to determine the efficacy of reviewing small pieces of code without their surrounding context, using Google Documents as our reviewing interface. The studies comprised 7 participants and were based on the WordFinder and RatPoly assignments. The results from those two studies indicated that while a certain class of problems require a complete functional unit of code to detect (design issues, etc.), our reviewers were still able to write a significant amount of useful and meaningful feedback on the small chunks of code they were given.

In Table 6.2, we show the number and size of chunks generated from each of the four assignments when processed with our partitioner but without chunk filtering. We see a large variation in chunk size on all assignments, and the only significant pattern that emerges is that WordFinder has, on average, larger chunks than the other assignments. This can be explained by the fact that WordFinder was an exercise in writing a Swing application in Java, which tends to produce longer methods. The other three assignments

	Chunk Count			Chunk Size (lines)			
Assignment	Total	Per Submission	Per Student	min	max	mean	stdev
WordFinder	2095	29.92	29.92	1	270	22.79	34.05
RatPoly	26330	230.96	230.96	2	443	13.28	30.39
Multipart	2393	59.82	19.94	1	194	15.77	16.85
Antibattleship	17653	534.94	178.31	1	723	13.47	27.88

Table 6.2: Pre-filtering chunk size distribution

	Chunk Count			Chunk Size (lines)			
Assignment	Total	Per Submission	Per Student	min	max	mean	stdev
WordFinder	801	11.44	11.44	6	270	27.46	34.03
RatPoly	2379	20.87	20.87	6	91	18.54	11.22
Multipart	1352	33.80	11.27	6	194	17.34	17.41
Antibattleship	9415	285.30	95.10	6	723	21.85	35.35

Table 6.3: Post-filtering chunk size distribution

either included small Swing components or none at all. We also explain the high number of lines per submission for RatPoly with the fact that we include comments when counting lines of code, and the template code distributed with RatPoly contained a large amount of method documentation. We use the number of chunks per *student* as a measure of the predicted review workload, since the students themselves will function as the primary reviewers in our system.

6.1.2 Chunk Filtering

After partitioning, we ran our filtering code on the chunks generated from our four sample assignments. The results for RatPoly are the most interesting, since the high number of submissions and large amount of staff-provided code generated an originally very large number of chunks. Our filters, especially the duplicate filter, however, managed to reduce the original set of 230.96 chunks per student to a much more manageable set of 20.87 chunks per student, less than 10% of the original input. The Antibattleship assignment here, by contrast, represents a worst-case input for our filters. The size filter managed to remove a moderate number of chunks, but the duplicate filter was essentially useless (removing less than 2% of the original chunks) since the assignment has no common template code. The resulting set of 285.30 chunks per submission, or 95.10 chunks per student (3 students per submission), is still admittedly too large to be reasonably reviewed, since we expect users in our system to review on the order of 10 chunks per assignment. Even if we consider that Antibattleship was a longer assignment, taking three weeks for implementation, the number of chunks generated per student does not compare favorably with the other assignments evaluated. We defer to future efforts to develop more sophisticated filtering approaches (see subsection 7.1.2).

Assignment	Original	Trivial	Duplicate	Final	Reduction
WordFinder	29.92	–6.67	–11.81	11.44	61.76%
RatPoly	230.96	–87.82	–122.28	20.87	90.96%
Multipart	59.82	–12.38	–13.65	33.80	43.50%
Antibattleship	534.94	–241.00	–8.64	285.30	46.67%

Table 6.4: Chunks removed per submission by preprocessor filters

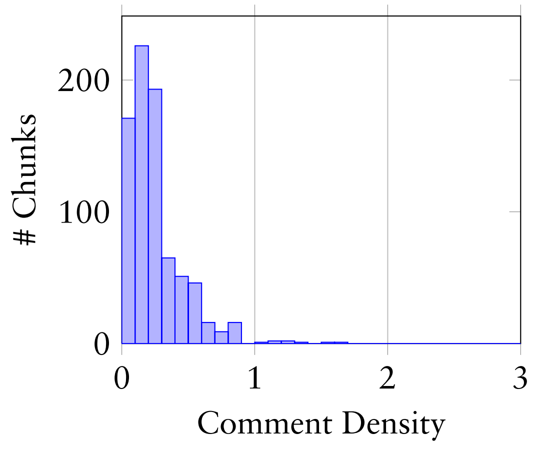
6.1.3 Automated Commenting

In Table 6.5, we show the number of comments generated by Checkstyle for each of the four assignments we evaluated the system against. Figure 6-1 shows the density of the generated Checkstyle comments as the number of comments per line of code for each chunk. We see that the distributions of comments for all four assignments are all roughly centered on the mean comment density for the entire assignment. The overall comment density is relatively high, ranging from 0.19–0.28 over our four assignments, even with all whitespace-related rules disabled. We argue, however, that our voting feature in the

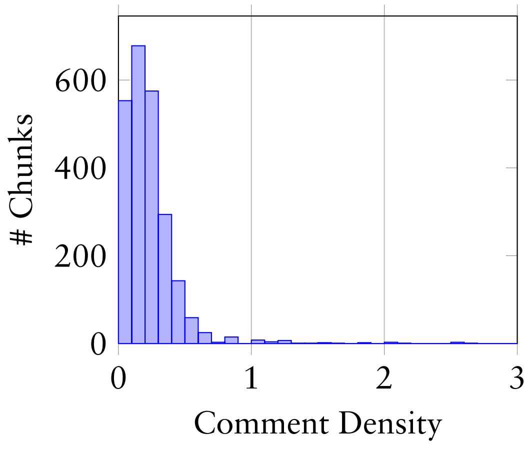
Assignment	Chunks	Comments	Comments / Line
WordFinder	801	4057	0.1921
RatPoly	2379	7981	0.1913
Multipart	1352	6358	0.2879
Antibattleship	9415	43610	0.2222

Table 6.5: Generated Checkstyle comment counts

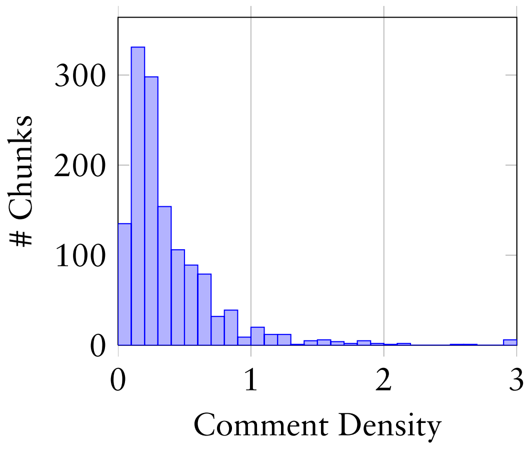
reviewing interface along with the ability to collapse comments at least partially alleviates this problem. In addition, in an actual deployment, the Checkstyle rules used by the system could be enforced outside of Caesar, reducing the number of violations that are submitted.



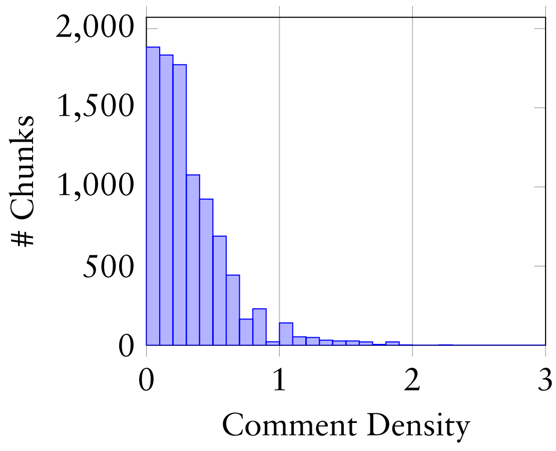
(a) 6.831 Problem Set: WordFinder



(b) 6.005 Problem Set: RatPoly



(c) 6.005 Project: Multipart

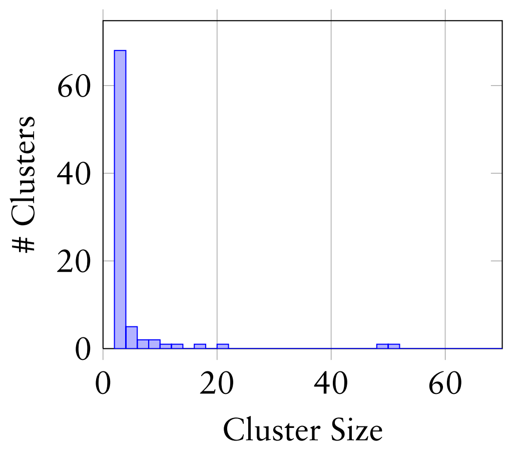


(d) 6.005 Project: Antibattleship

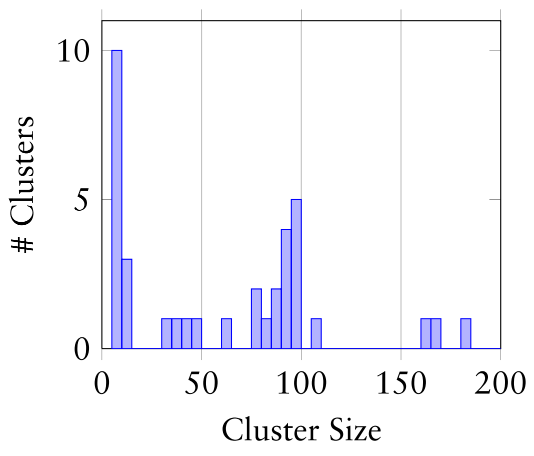
Figure 6-1: Checkstyle comment density distributions

6.1.4 Clustering

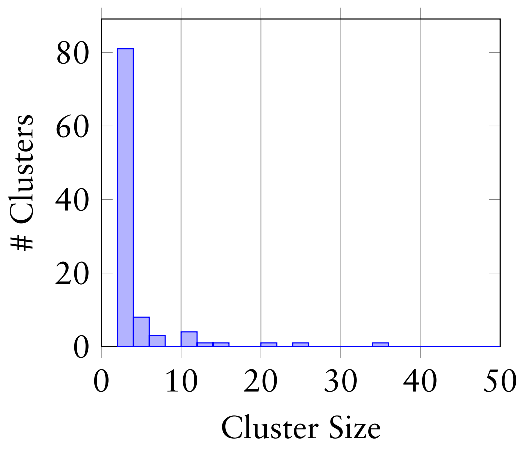
We evaluated our clustering implementation on the same four assignments. Our cluster distance threshold was set to 0.95 and our name penalty was $k_{np} = 0.7$. These parameters were tuned so that chunks that share a name need very few fingerprints in common to be considered similar, while chunks that do not share a name need to share a large proportion of their fingerprints.



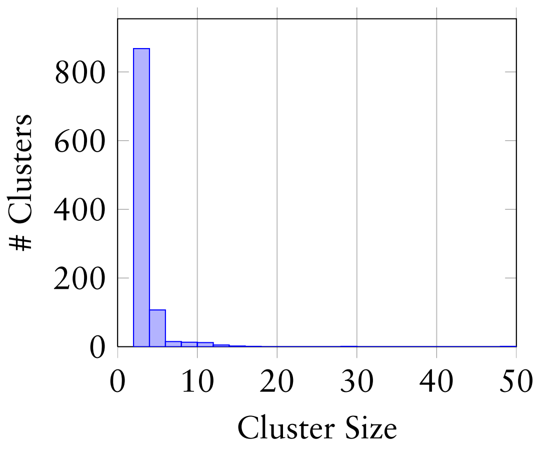
(a) WordFinder (70 submissions)



(b) RatPoly (114 submissions)



(c) Multipart (40 submissions)



(d) Antibattleship (33 submissions)

Figure 6-2: Size distribution of clusters of 2 or more chunks

Figure 6-2 shows the cluster size distributions for the four assignments. Notably, we see a few large clusters in WordFinder and Multipart where students have modified staff-provided template code, with the rest of the chunks falling into either small clusters or no cluster at all. Since both assignments were structured to have a small staff-provided interface template and a large proportion of student-structured implementation, these distributions are reasonable. The RatPoly assignment, because it is almost entirely composed of staff-written templates that students simply fill in, generated relatively more large clusters. The sizes of most of these large clusters corresponds favorably with the number of submissions (114). The three outliers with more than 150 chunks can be explained by the clustering algorithm collapsing clusters of very similar methods, like `RatPoly.add(..)` and `RatPoly.sub(..)`. Finally, Antibattleship does not yield any large clusters. Presumably this is due to the fact that the assignment was only loosely specified, and no common structure was enforced on the students. A more sophisticated clustering technique could have potentially produced better results (see subsection 7.1.3), since our fingerprint-based approach is susceptible to variations in variable names, statement order, etc.

6.2 Task Routing

We evaluated our task routing system using a combination of real chunk data from the 4 assignments used throughout our evaluation and simulated users modeled after statistical characteristics of a hypothetical class. For each assignment, we assumed that the number of students was exactly determined by the number of submissions, and then assumed one teaching staff member for every 20 students, and one alumnus for every 2 students. Next, in order to accurately model user reputation, we assigned each user a reputation value sampled from a Pareto distribution (long tail) with $\alpha = 1.2$, which we chose to emulate the reputation characteristics of sites like Stack Overflow. We also gave staff members constant 20 point reputation boost. Students were assigned 8 chunks each, staff were assigned 28 chunks each, and other users were assigned 5 chunks each. We assumed that user arrival time into the system is random, except that teaching staff enter the system after all other users. The task router was configured with a goal of 2 reviewers per chunk. When assigning users chunks from a cluster, we set a maximum of 3 chunks per cluster to assign to a single user, preventing the system from assigning one user all chunks exclusively from a single cluster and risking losing reviewer interest.

The results of our simulated task routing are summarized in Table 6.6 and Table 6.7. For our purposes, we will say that when two users are both assigned to review the same chunk, they are *co-reviewers*. Table 6.6 shows how many users from each role, on average, are co-reviewers with each student. These numbers are important for measuring the diversity of reviewing experience for students, and also for ensuring that all students have a fair amount of co-reviewer interaction with staff. We see that students in general have a reasonable amount of staff interaction as reviewers. In fact, in WordFinder there were 58 out of 70 students who had at least one staff co-reviewer. For RatPoly, there were 100 out of 114, for Multipart there were 94 out of 120, and for Antibattleship there were 81 out of 99. These numbers are especially impressive considering that our task routing algorithm prioritizes staff allocation by submission, not by co-reviewer interaction.

	Student		Staff		Other	
Assignment	mean	stdev	mean	stdev	mean	stdev
WordFinder	2.77	0.56	1.83	1.08	0.61	0.76
RatPoly	2.75	0.54	2.42	1.46	0.64	0.71
Multipart	2.53	0.71	2.79	2.06	0.62	0.67
Antibattleship	2.84	0.85	1.78	1.29	0.66	0.77

Table 6.6: Task assignment student co-reviewer interactions

Table 6.7 summarizes the performance of our task routing algorithm by submission using two metrics: number of distinct reviewers per submission, and the percentage of chunks with at least one reviewer per submission. It is notable as well that for all assignments, there were no submissions that did not have at least one staff reviewer assigned. The high standard deviations for chunk coverage for WordFinder, Multipart, and Antibattleship can be explained by the fact that our approach tries to ensure that an even number of chunks per submission are reviewed. These three assignments do not impose a structure on student submissions, and therefore see a larger variation in the number of chunks per submission. It follows that chunk coverage would exhibit a correspondingly high variation.

	Distinct Reviewers				Chunk Coverage (%)			
Assignment	min	max	mean	stdev	min	max	mean	stdev
WordFinder	5	48	10.97	6.87	14.29	100.00	62.51	25.24
RatPoly	5	16	13.32	1.92	9.46	50.00	26.61	5.24
Multipart	16	40	35.24	3.65	20.22	100.00	53.37	19.78
Antibattleship	2	118	16.88	20.44	0.61	11.88	4.33	3.34

Table 6.7: Task assignment metrics by submission

6.3 Reviewing Interface

To evaluate the usability of our reviewing interface, as well as the experience of reviewing small chunks of code without their surrounding context, we ran a user study with 18 users and submissions drawn from the WordFinder assignment. Participants were asked to review 3 chunks (methods) each by voting on or replying to existing comments, or writing new comments. After, they were asked to complete a short survey about their experience with the interface. Each participant wrote an average of 2.61 comments, of which 21.28% were replies to other comments, and contributed an average of 3.50 votes.

Response to the interface was generally positive, with several users praising the interface and no users expressing any serious problems with it. 7 out of 18 of the participants, however, indicated that they felt like they were not presented enough information or context to effectively review the code they were assigned. While some of this can be attributed to a lack of enough background information for the reviewers, it also identifies one of the areas where Caesar could be most improved (see subsection 7.1.1). We have partially addressed the lack of context surrounding a chunk by allowing users to view the code for the entire original submission (subsection 5.2.3). Two of the participants expressed that they liked the voting feature as a lightweight mechanism for expressing either agreement or disagreement. Finally, user sentiment on the automated Checkstyle comments being presented along with the reviewing interface was mixed. A few users disagreed with some of the rules that Checkstyle was enforcing, an issue that would be resolved in a classroom setting with an agreed upon coding style. Others wrote that they appreciated having a tool for automatically finding obvious or mechanical issues instead of forcing users to do the same manually, validating our original design goal for including automated comments.

Chapter 7

Conclusion

Caesar is a distributed, social code review system developed for programming education that allows students to receive faster and better feedback on their work in addition to experience reviewing code, reduces the grading load on teaching staff, and enables alumni to interact with students in a meaningful and mutually beneficial way. While previous work has attempted to build a useful code review tool for the classroom, none of them have adequately translated the code experience of code review as it exists in industry while simultaneously designing for the unique constraints of a large programming class of students.

The classroom presents a number of design constraints not found in professional practice. The possiblity of plagiarism and students' lack of experience with version control precludes the option of reviewing individual commits, which motivates our decision to implement a content partitioning scheme for submissions. Dividing code into chunks coupled with filtering and task routing also gives us the ability to scale past standard programming team sizes of a handful of expert developers up to a classroom of over 100 students of varying level and ability. In addition, reviewing student submissions for an assignment could potentially mean reviewing many redundant but different implementations of the same specification, a pattern which we exploit using code similarity clustering. It is also important, in the classroom, to evenly allocate teaching resources between students, and ensure as much as possible that students all receive the same learning opportunities, a goal we address with our task routing strategy.

The results from evaluating our code preprocessing pipeline and task routing algorithm on four real assignments from previous iterations of 6.005 and 6.813 both validated our system design against assignments like RatPoly with a large amount of common template code, and highlighted its inability to cope with large and unstructured projects like Antibattleship. While part of this is simply due to the relatively large amount of code produced in a large project assignment, it is our belief that future efforts could implement code summarization techniques and more sophisticated filtering to at least partially address the issue. Finally, A user study of our reviewing interface yielded mostly positive feedback on its usability, but indicated that reviewers wanted access to more context about the code they were reviewing.

Caesar makes the following contributions:

- A code review system specifically for classroom use
- A distributed, crowd-sourced code reviewing workflow
- A streamlined, social web interface for reviewing code

Beyond these, though, Caesar provides an extensible, modular framework that decomposes the process of code review into a powerful pipelined architecture composed of a partitioner, chunk filter, static analysis, similar code clusterer, task router, and reviewing interface. Our current implementation provides simple but adequate implementations of all of these pieces of the system, but it is our belief that Caesar only lays the foundation for future work to improve on its individual components.

7.1 Future Work

7.1.1 Additional Forms of Review Content

Our preprocessing system implements a very simple notion of what constitutes a chunk of code. In the future, we would like to pursue more advanced techniques for providing reviewers with the best possible content to review.

Most intriguing, perhaps, is the idea of *summarizing* code instead of simply partitioning it. If designed correctly, a code summarization strategy could allow for reviewers to make comments about high-level aspects of student submissions without having to read through all of the code themselves. One can imagine a scheme where reviewers examine only the names of the classes written by a student and where they are located in the package tree, or perhaps they are given only a class dependency diagram generated from student projects.

Another unexplored area is the possibility of including information generated dynamically from submitted code, instead of information obtained through static analysis. It is very likely that examining execution traces from a program could reveal certain types of defects more easily than examining the code itself in a static context. We could also consider including results from unit tests, code coverage analysis, or performance profiling information.

7.1.2 Improved Chunk Filtering

In addition to generate different forms of reviewable content, there are also several promising areas of improvement for filtering out less relevant content before uploading to the review interface. While we currently filter out commonly repeated (most likely staff-provided) chunks and trivial chunks, the resulting set of chunks is still quite large for even modest assignments. Moving beyond these simple approaches, we would like to investigate using other heuristics for determining what content to review, such as:

Churn

Currently our system ignores submission commit histories. We could potentially mine the commit history to find the areas of code that experienced the largest number of changes. Our reasoning is that areas that experience more development activity are likely to be error-prone, complex, or important, and therefore good candidates for review.

Complexity

We would also like to investigate the possibility of identifying complex areas of code as candidates for review. This would most likely have to be implemented as a language-specific heuristic approach, using traits such as identifier length, conditional branching depth, number of dependencies.

Coverage

Looking at the execution of submitted code, we could also examine regions of dead code, or conversely hot regions that are executed more often than average.

7.1.3 Improved Chunk Clustering

Many of the ideas for more content-aware chunk filtering can also be applied to improving our similar chunk clustering algorithm. While using document fingerprints allows for some degree of tolerance in detecting similar chunks, its language-agnostic nature prevents it from detecting syntactically or semantically equivalent code that differs in content such as variable names or other identifiers. Implementing a more sophisticated approach like the one described in [9, 17], or one of the techiques described in one of the many surveys of code clustering or clone detection techniques [27, 28] could yield better results.

7.1.4 Data-Driven Task Routing

Our task routing implementation currently looks at existing reviewing assignments and similarity clusters in a simple ranking scheme. A better approach would be to design a more data-driven routing algorithm that can adaptively incorporate feedback from the reviewing process to generate better routing assignments. It could do this by constructing user models for reviewers based on their actions in the system, adding more knowledge about the contents of chunks beyond just similarity clusters, and moving to more advanced approaches drawing on techniques like collaborative filtering.

7.1.5 Similar Comment Clustering

As a system for gathering user annotations on student code, Caesar presents an opportunity to analyze the comments written by reviewers and use them to generate aggregate information about the reviewed code. By identifying comments that indicate the same problem, we could potentially generate implicit clusters of code segments that share the same bug. It would be tremendously useful as a tool for instructors to be able to retrieve all instances of representation exposure defects found in student code, or perhaps a listing of all types of infinite loops written by students. By adding some form of structure to the commenting system, we could potentially use code review as a defect labeling system without adding unnecessary burden on our users.

7.1.6 Community Building

Caesar's web interface has a few features that together constitute basic support for a community-driven code review site. While it is outside the scope of this thesis to explore the ramifications and possibilities of building a community in a code reviewing context, it is our belief that there are major gains to be made in the effectiveness of code review by investigating ideas like achievement badges, allowing users to follow or friend other users, integrating a messaging system, and improving our activity stream implementation.

Bibliography

[1] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: a word processor with a crowd inside," *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, 313–322, 2010, ACM ID: 1866078. DOI: 10.1145/1866029.1866078.

[2] *Checkstyle*. [Online]. Available: http://checkstyle.sourceforge.net/.

[3] M. Chu-Carroll. (Jul. 2011). Things everyone should do: code review, [Online]. Available: http://scientopia.org/blogs/goodmath/2011/07/06/things-everyone-should-do-code-review/.

[4] (Apr. 1999). Code conventions for the Java programming language, [Online]. Available: http://www.oracle.com/technetwork/java/codeconv-138413.html.

[5] J. Cohen, "Code review at Cisco systems," in *Best Kept Secrets of Peer Code Review*, Smartbearsoftware.com, 2006, pp. 63–87. [Online]. Available: http://smartbear.com/resources/cc/book/code-review-cisco-case-study.pdf.

[6] D. Cosley, D. Frankowski, L. Terveen, and J. Riedl, "SuggestBot: using intelligent task routing to help people find work in Wikipedia," *Proceedings of the 12th international conference on Intelligent user interfaces*, IUI '07, 32–41, 2007, ACM ID: 1216309. DOI: 10.1145/1216295.1216309.

[7] R. M. Crespo, A. Pardo, J. P. S. Pérez, and C. D. Kloos, "An algorithm for peer review matching using student profiles based on fuzzy classification and genetic algorithms," *Proceedings of the 18th international conference on Innovations in Applied Artificial Intelligence*, IEA/AIE'2005, 685–694, 2005, ACM ID: 1117011. DOI: 10.1007/11504894_95.

[8] *Crucible*. [Online]. Available: http://www.atlassian.com/software/crucible/.

[9] W. S Evans, C. W Fraser, and F. Ma, "Clone detection via structural abstraction," *Software Quality Journal*, vol. 17, no. 4, 309–330, 2009.

[10] M. E. Fagan, "Advances in software inspections," *IEEE Transactions on Software Engineering*, vol. 12, 744–751, Jul. 1986, ACM ID: 9778, ISSN: 0098-5589. [Online]. Available: http://portal.acm.org/citation.cfm?id=9775.9778.

[11] E. F. Gehringer, "Electronic peer review and peer grading in computer science courses," *ACM SIGCSE Bulletin*, SIGCSE '01, 139–143, 2001, ACM ID: 364564. DOI: 10.1145/364447.364564.

[12] *Gerrit*. [Online]. Available: `http://code.google.com/p/gerrit/`.

[13] *Hac: hierarchical agglomerative clustering library*. [Online]. Available: `https://github.com/sape/hac`.

[14] J. Howe, "The rise of crowdsourcing," *Wired*, no. 14.06, Jun. 2006. [Online]. Available: `http://www.wired.com/wired/archive/14.06/crowds.html`.

[15] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan, "Integrating pedagogical code reviews into a CS 1 course: an empirical study," *ACM SIGCSE Bulletin*, vol. 41, 291–295, Mar. 2009, ACM ID: 1508972, ISSN: 0097-8418. DOI: `10.1145/1539024.1508972`.

[16] C. Hundhausen, A. Agrawal, and K. Ryan, "The design of an online environment to support pedagogical code reviews," *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, 182–186, 2010, ACM ID: 1734324. DOI: `10.1145/1734263.1734324`.

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 654–670, 2002. DOI: `10.1109/TSE.2002.1019480`.

[18] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "TurKit: tools for iterative tasks on Mechanical Turk," *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, 29–30, 2009, ACM ID: 1600159. DOI: `10.1145/1600150.1600159`.

[19] E. Z. Liu, S. S. Lin, and S. M. Yuan, "Alternatives to instructor assessment: a case study of comparing self and peer assessment with instructor assessment under a networked innovative assessment procedures," *International Journal of Instructional Media*, vol. 29, no. 4, 395–404, 2002.

[20] N. F Liu and D. R. Carless, "Peer feedback: the learning element of peer assessment," 2006.

[21] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, 2857–2866, 2011, ACM ID: 1979366. DOI: `10.1145/1978942.1979366`.

[22] *Mondrian*. [Online]. Available: `http://www.youtube.com/watch?v=sMql3Di4Kgc`.

[23] K. Reily, P. L. Finnerty, and L. Terveen, "Two peers are better than one: aggregating peer reviews for computing assignments is surprisingly accurate," *Proceedings of the ACM 2009 international conference on Supporting group work*, GROUP '09, 115–124, 2009, ACM ID: 1531692. DOI: `10.1145/1531674.1531692`.

[24] *Review Board*. [Online]. Available: `http://www.reviewboard.org/`.

[25] *Rietveld*. [Online]. Available: `http://code.google.com/appengine/articles/rietveld.html`.

[26] P. C. Rigby and M. Storey, "Understanding broadcast based peer review on open source software projects," *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, 541–550, 2011, ACM ID: 1985867. DOI: 10.1145/1985793.1985867.

[27] C. K Roy and J. R Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, p. 115, 2007.

[28] C. K Roy, J. R Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: a qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, 470–495, 2009.

[29] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, 76–85, 2003, ACM ID: 872770. DOI: 10.1145/872757.872770.

[30] D. Sluijsmans, F. Dochy, and G. Moerkerke, "Creating a learning environment by using self-, peer-and co-assessment," *Learning Environments Research*, vol. 1, no. 3, 293–319, 1998.

[31] A. Trivedi, D. C. Kar, and H. Patterson-McNeill, "Automatic assignment management and peer evaluation," *Journal of Computing Sciences in Colleges*, vol. 18, 30–37, Apr. 2003, ACM ID: 767605, ISSN: 1937-4771. [Online]. Available: http://portal.acm.org/citation.cfm?id=767598.767605.

[32] D. A. Trytten, "A design for team peer code review," *ACM SIGCSE Bulletin*, SIGCSE '05, 455–459, 2005, ACM ID: 1047492. DOI: 10.1145/1047344.1047492.

[33] S. Turner and M. A. P. nones, "Exploring peer review in the computer science classroom," *CoRR*, vol. abs/0907.3456, Jul. 2009. [Online]. Available: http://arxiv.org/abs/0907.3456v1.

[34] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, 1st ed. Addison-Wesley Professional, Nov. 2001, ISBN: 0201734850.