A Mobile Instructor Interface for Collaborative Software Development Education

by

Angela N. Chang

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

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 2012

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

Certified by . . . .
Robert C. Miller
Associate Professor of Computer Science and Engineering
Thesis Supervisor
May 21, 2012

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A Mobile Instructor Interface for Collaborative Software Development Education

by

Angela N. Chang

Submitted to the

Department of Electrical Engineering and Computer Science

May 21, 2012

In partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

Students are often asked to write code during lab sessions in software engineering courses. However, the overall progress and level of understanding of lecture material during the course of a single lab session is difficult for instructors to gauge, because they are limited in the amount of direct interaction they can have with students. We have built CollabodeTA, a web application optimized for Apple's iPad on top of the Collabode real-time collaborative web IDE. CollabodeTA is a tool that takes advantage of keystroke-by-keystroke and action-by-action data intercepted through Collabode to aid software lab instructors in determining student progress and understanding on in-class coding assignments. User studies using TAs from MIT's 6.005 Elements of Software Construction course and data recorded from a semester of 6.005 recitations with in-class coding assignments indicate that the mobile instructor interface shows potential as a useful tool for guiding the pace and content of such recitations based on demonstrated student understanding. Furthermore, the CollabodeTA mobile instructor interface illustrates a new use case for the Collabode real-time collaborative web IDE.

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

Acknowledgments

There are many individuals whose support helped this thesis become a reality. First, I would like to thank my advisor Rob Miller, whose zeal for software and user interface design I admired even before joining the User Interface Design (UID) group at MIT CSAIL, and whose vision, advice, and leadership carried me throughout the thesis process. He is the best listener I have ever met, pushed me to become a better writer and researcher, and was a constant source of support, ideas, and wisdom as well as the force that kept me making consistent progress on this work throughout the year.

Collabode is the brainchild of Max Goldman, and I couldn't have gotten luckier when it comes to working with someone who is so technically capable, driven about doing research that will make a difference in the lives of others, and fun to work with all at the same time. In developing this thesis, I have learned the most about software engineering from him, and would like to thank Max for letting me take the seed of his mobile instructor idea and start to make it grow.

Mason Tang was without a doubt my most avid supporter, confidante, and friend throughout this entire process, both within the context of developing the body of work behind this thesis and managing in the many threads of my life outside of this work that constantly fought for my time and attention. I am deeply grateful for his insights, suggestions, references, and reassurances throughout the year that kept me strong and moving forward.

Similarly, David Chen was a constant fountain of support. In particular, the console output diff view that started out looking very different yet became so key to CollabodeTA bloomed after discussing it with him. I'd also like to thank my roommates Drew Wolpert, Anna Fung, and friend Ben Lee for helping me through and putting up with countless rants, brainstorming sessions, and a year full of bizarre working schedule.

Without the data from the Fall 2011 6.005 TA recitations, the evaluation for this thesis would not have been possible. I am forever grateful especially to Elena Tatarchenko, Samuel Wang, and Amelia Arbisser for taking the extra time to help me evaluate CollabodeTA a whole semester after teaching the class, and for providing valuable insight that I could not have gotten anywhere else.

Thanks go to the members of the User Interface Design group at MIT CSAIL for always being curious and enthusiastic, and for giving me confidence in and great feedback on my project whenever we discussed it. I wish I could have worked with them more!

Finally, I would like to thank my family, for whom this thesis is written. I would not be where I am today without them.

Table of Contents

Chapters

List of Figures

Chapter 1

Introduction

Software engineering and programming courses are becoming increasingly and undeniably popular in modern-day universities. Because it is difficult to teach programming or software engineering without hands-on experience, most programming and software courses incorporate in-class coding opportunities in the form of software labs, recitations, or office hours, where students can work on code under the supervision of one or more instructors who are there to teach and answer student questions. In software engineering courses in particular, it is common in such classes to have students working together on larger-scale assignments, both to enforce principles taught in class and to give students experience with collaborative software development tools used in the workplace.

However, in our experience, the software lab situation is more opaque to instructors than it could be. In a typical lab or recitation setting, students code by themselves or in small groups, and instructors know very little about what the majority of the class is doing at any given point in time. Often, they have to wait to be called over by a student with a question and spend some amount of time looking over the student's code with them before they can give the help the student needs. Additionally, without constantly polling the class and hoping for honest responses each time, there is no way for an instructor to know how accurately how well students are completing the lab exercises and understanding key concepts, and which topics or pitfalls need to be reiterated to the class as a whole. This problem is especially apparent in the common situation where multiple students are having the same difficulty and instructors are only working with one or a few at a time.

With new collaborative software development technologies, we believe that this barrier between instructors and their students should no longer be the norm.

Collabode is a real-time web-based integrated development environment (IDE) created as a platform for collaborative software development [1]. The introduction of Collabode to the world of software development enables new scenarios for close, synchronous collaboration between two or more programmers actively contributing to the same body of code. The original papers [2, 3] describe three such models of collaboration: test-driven pair programming, micro-outsourcing, and mobile instructor.

Prior to the work presented in this thesis, two of the three models had been studied through user evaluations of the Collabode system. The goal of this thesis was to design and build an interface supporting the third collaboration model, the "mobile instructor". This model makes use of Collabode in a classroom software lab setting, where one or more lab instructors assign coding assignments to classes of 30-100 students. Instructors are responsible for monitoring student progress and understanding by answering questions and providing suggestions throughout the lab session. The mobile instructor interface is therefore an interface that will be used by lab instructors with the goal of increasing the quality and efficiency of student/teacher interaction and software engineering education.

We have built CollabodeTA, a web application optimized for the Apple iPad on top of the Collabode real-time collaborative web IDE that uses keystroke-by-keystroke and action-by-action data input by students and intercepted through Collabode to provide a tool to aid software lab instructors in determining student progress and understanding during in-class coding assignments. This augments previous work in the area of collaborative coding, web IDEs, and technology-enabled classroom tools by taking advantage of the information that collaborative, real-time web IDEs provide and applying it to the area of software development education. The data we used to both design and evaluate CollabodeTA came from MIT's 6.005 Elements of Software Construction course, a popular introductory class to software development, design principles, and tools, taught in the Java programming language. Assignments in 6.005 (as this course will be referred to for the rest of this thesis) consist of software design and implementation projects completed individually and in small groups, as well as programming exercises assigned in hour-long recitations of 15-30 students.

Our implementation of CollabodeTA consists of a collection of views and metrics meant to be viewed on a touch tablet such as the Apple iPad. We aimed to accomplish the following goals:

- Provide an at-a-glance summary of student activity requiring minimal user interaction and interruption of an instructor's accustomed workflow while incorporating useful information in a concise way
- Help lab instructors gauge student understanding during the course of a lab session based on student work and demonstrated understanding while completing assignments, in order to guide the pace of their teaching and adjust on the fly if necessary
- Help lab instructors identify student outliers who either might need additional, proactive help from the instructor, or who demonstrate exceptional understanding

of the given assignment and in which case can be asked to help their fellow classmates or whose code can be used as a teaching example

Within CollabodeTA, each student's information is concisely summarized on a *student card*, which contains a picture of the student, a label with their Collabode username, and badges indicating student status according metrics we collect. Our implementation includes a badge indicating a student's help queue status, and other badges can easily be added.

While we tried to make CollabodeTA as glanceable as possible, we also provide a few different views with which instructors can monitor class progress. These views are as follows:

Class layout view

The class layout view is a visual summary of student progress. Within the class layout view, all student cards are visible at once, potentially providing the most complete overview of student performance by many different metrics. Student cards in this view can be rearranged to match actual locations of students in the classroom, to help instructors locate specific students.

Console output view

In our evaluations of recitations from 6.005, we found analyzing console output to be one of the richest sources of information about student progress. From looking at console output, we could gather which exercises students were working on at any given time, what errors they got and how they compared to the other students in the class, which tests they passed or did not pass, and so on. Therefore, the console output view exists as a way for instructors to compare console output from student code in real time, in order to discern information such as class progress on assignments and common errors.

Student panel view

Every student card has an associated student panel, which is a view showing detailed information about that single student. It is intended to provide additional important information to instructors who are helping or preparing to help a specific student when it is needed, and hides details about other unhelped students that would otherwise clutter up the summary views.

Help queue

A simple help queue can be shown or hidden to the left of any of the informational views. This allows us to organize students asking for instructor help in a first in, first out order.

Figure 1-1 shows the CollabodeTA interface, which is further described in Chapter 3.

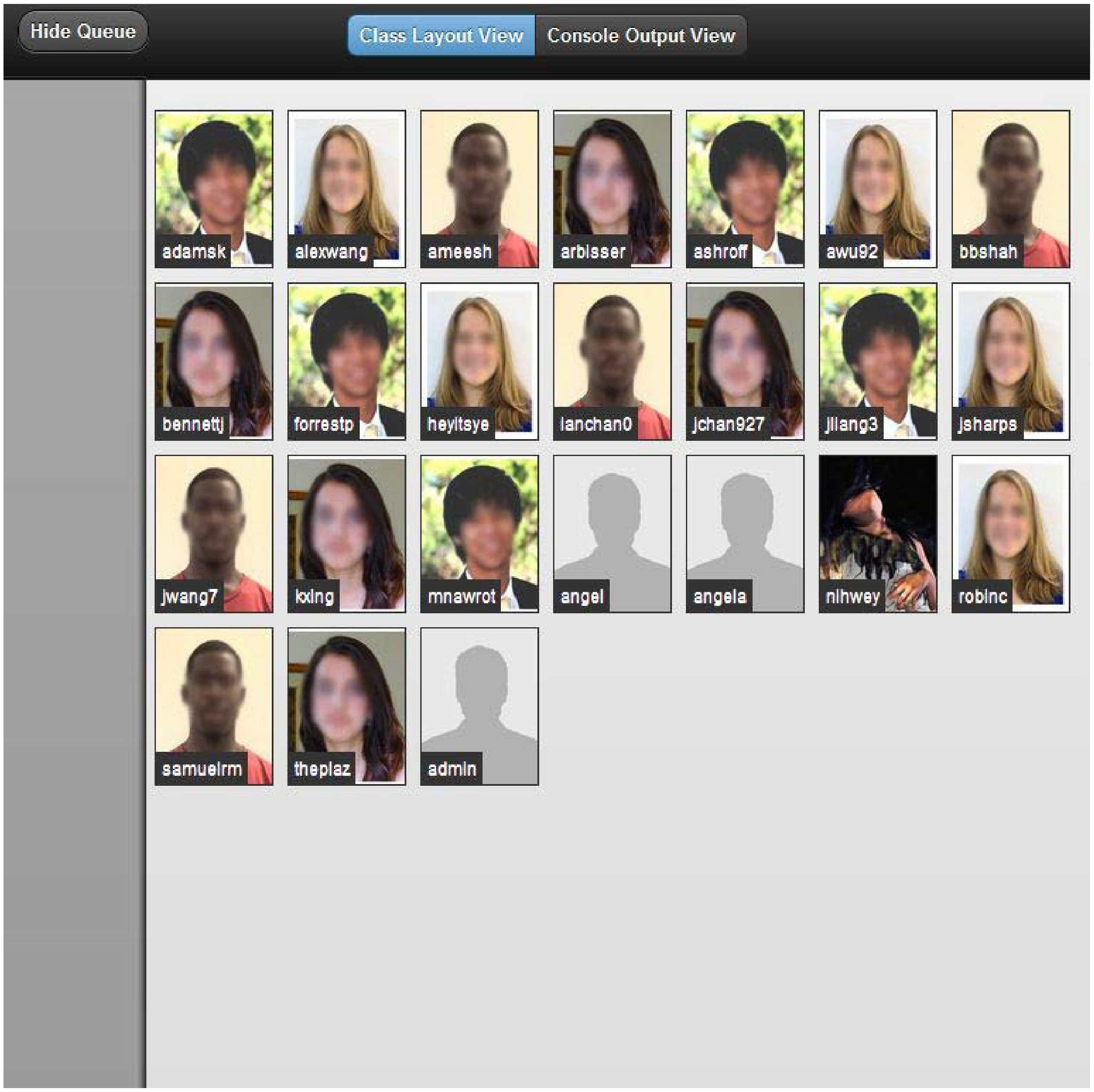


Figure 1-1. *CollabodeTA mobile instructor interface.*

Evaluation of the mobile instructor interface was performed using both teaching assistants and data from the Fall 2011 semester of 6.005, where students used Collabode to complete coding assignments in recitations. In order to perform this evaluation, a replay system was built to play back recorded recitation data and simulate the use of CollabodeTA in a live classroom.

The aim of this thesis is twofold: First, to evaluate the effectiveness of the mobile instructor collaboration model, and second, to describe the Collabode mobile instructor interface, which demonstrates new possibilities for software development education enabled by code collaboration in the classroom.

The remainder of this paper presents the context, design, implementation, and evaluation of the mobile instructor interface. Chapter 2 introduces the problems Collabode was created to solve, briefly describes the Collabode system itself, and summarizes related work. Chapter 3 discusses the motivations, design, and implementation of each aspect of the mobile instructor interface. Results from our evaluations of CollabodeTA through user studies and post-recitation analysis are presented in Chapter 4. Finally, Chapter 5 presents a conclusion and discussion of future work.

Chapter 2

Background

2.1 Interactions in Software Labs

MIT has several programming and software development classes during which students complete coding assignments in class under the supervision of lab instructors (primary lecturers), TAs (teaching assistants), and LAs (student lab assistants). The following observations of how students interacted with instructors during such classes were made during Fall and Winter 2011 in several programming labs:

- 6.005 recitations (15-30 students each, with 1 TA)
- 6.092, an introductory Java lab class (about 200+ students, with 3 instructors)
- 6.189, an introductory Python lab (about 150 students each, with 1-2 TAs and 3-5 LAs)
- A big data class (about 50 students, with 2 instructors)
- Office hours for 6.00, an introductory Python class (about 20 students each, with 3-4 TAs or LAs)

We observed that students receive help in either an active or a passive manner. Students who *actively* request help from the instructor do so by catching the instructor's attention. The manner in which this was done varied depending on the size of the class. In the introductory Java lab, which had about 200 students in a lecture hall, the class help queue consisted of students physically lining up to talk to one of three instructors. In the other smaller classes of 15-50 students with one or two instructors, students either raised their hands or simply shouted out their question to catch the instructor's attention. Each class we observed went through a peak period where many students actively requested help on the in-class assignment.

Passive instruction is given when there are lulls in active student requests, or when students feel that they are confused about some part of the assignment but do not feel like they are justified in asking for help, either because they feel they have not made enough progress, cannot clearly state what their problem is to the instructor, or that their question is "dumb" and they have not yet reached a base level of understanding required to complete the assignment. In all but the largest class we observed, when students did not actively engage instructors, instructors began to cycle through the class, looking over students' shoulders and picking out students to ask if they had any questions. We realized that especially when a student's motivation for staying quiet is because of a lack of understanding of the assignment, this somewhat arbitrary and passive process of receiving help often overlooks those most in need of instructor attention.

When actually giving help, instructors preferred being close to students so that they could see the student's code, ask them questions about their approach, and view the results of running student code. After seeing enough student implementations, instructors sometimes asked for the attention of the entire class to clarify common problems and mistakes. However, this required a single instructor to have seen enough students with the same mistake, which is more difficult to discern when multiple instructors are helping different students in the same class, and often did not become apparent until a significant portion of the lab time had elapsed and a significant portion of the class had moved past the problem in question.

With CollabodeTA, we hope to give instructors a more clear view of which students may be passively waiting for help, and to identify common mistakes early enough to address them without having to discover them through what may be numerous lengthy direct interactions with students.

2.2 The Collaboration Problem

The problem of collaboration in software development is detailed in the published Collabode papers [2, 3], and is the key motivation for Collabode, the system upon which CollabodeTA is built. Collabode was built in part to address the inadequacies of the two major options programmers have to collaboratively write code, which are pair programming and version control. Furthermore, within the area of software engineering or programming education, we believe that both pair programming and version control models of collaboration create undesirable barriers, not only between student collaborators on code, but also between programmers (students) and observers of the code and coding process (instructors).

Pair programming is the practice in which two (or more) programmers share a computer screen, keyboard, and mouse. All programmers follow along and contribute to the code's progress as one designated programmer does the work of typing actual changes into the code. Despite sharing a view of the screen, only one programmer has control over what gets entered into the computer. In practice, even if each coder in a pair programming situation is assigned a specific role (for example, if one is designated to code and the other to catch potential bugs and typos), there is little hope for both coders to think on substantially different levels of abstraction when actively discussing and looking at the same small section of code at a time. In a classroom or software lab context, where students often think at different rates and need time to absorb class concepts and practice writing code, this creates an unequal learning experience and creates the potential for students with weaker understanding of the subject material to hide behind those with stronger abilities. From a lab instructor's point of view, it is often impossible to tell whether or not contributions are being made equally in a team using a pair programming strategy.

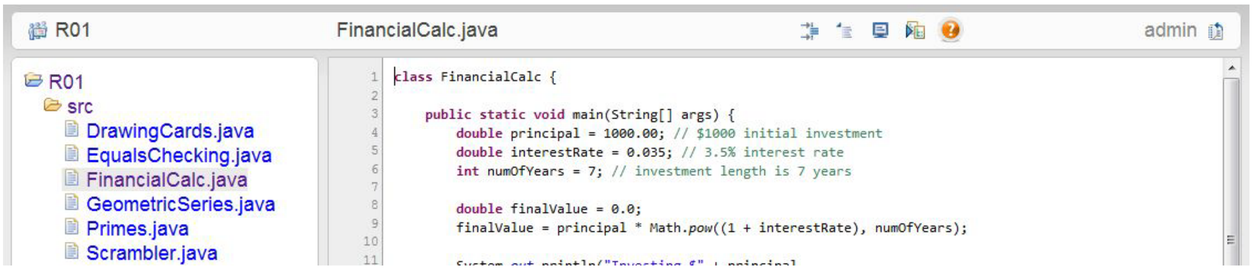
The *version control* model of collaboration allows programmers to work separately, at their own pace within the constraints of the team's overall deadlines. Working on separate tasks is encouraged, as conflicts that occur when code from multiple contributors is committed into the source repository must be manually reconciled. Again, we can identify problems with this model, particularly when implemented in a classroom lab setting. While version control allows multiple programmers to develop code in parallel, all in-progress code is completely hidden until both the author of the code decides to commit and push it to the central repository, *and* the other contributors interrupt their workflow to pull in the newest changes. In addition, collaborators must make an effort to keep their code conflict-free when editing the same or closely-related bodies of code, as a single conflicting update could potentially block all progress until it has been properly merged with new code. For students as well as for lab instructors, it is often helpful to observe the code development and thought process of other students in the class, and the version control model makes this process opaque. Furthermore, in a timed lab setting, minimizing the time spent merging code conflicts increases the time students can spend actually solving the assignment.

Variations of these two collaboration methodologies exist, but we believe that existing options boil down to these two core scenarios. As we can see, there are problems which show that neither of these scenarios is ideal for the type of collaboration in which multiple programmers wish to contribute actively and synchronously to the same body of code, especially in an educational setting. Furthermore, when students are asked to work on assignments in class, which implies working either individually or under one of these two models of collaboration, there is no satisfactory solution for instructors to watch over the programming process.

2.3 Collabode

Collabode aims to solve the collaboration problem by providing a web-based IDE allowing multiple programmers to work on separate machines while each having their own view of the complete code source, which is updated in near real time. The editor is based on Etherpad and is implemented in HTML and Javascript, which allows multiple collaborators to simultaneously access the IDE and modify code through a standard web browser simply by visiting a shared URL. Projects are hosted on a shared Collabode server, which uses Eclipse both to manage the projects and also to provide much of the IDE's core functionality such as syntax highlighting, continuous code compilation, compiler errors and warnings, and code formatting, refactoring and execution. Currently, Collabode supports these features for Java programs and allows printing output to the user's console. Programs with graphical user interfaces are not supported.

Collabode uses a technique called error-mediated integration to integrate only code without compilation errors into the main code body on disk, which allows programmers to make their own changes to the code base and see changes made by others as those changes are being made, in real time, without worrying about merge conflicts. It allows for much closer collaboration on code design and implementation decisions regardless of whether the programmers are co-located, and user studies have shown Collabode to significantly and positively impact the collaborative coding situation.



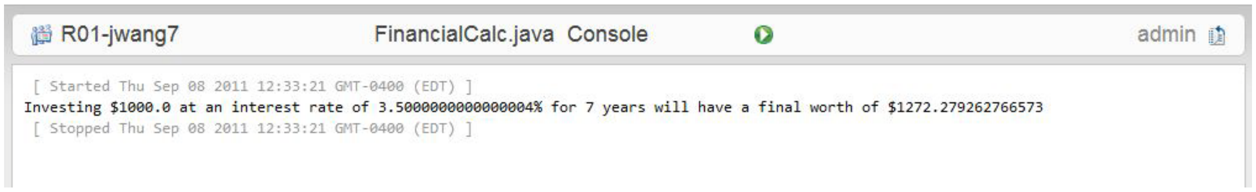
```
class FinancialCalc {

    public static void main(String[] args) {
        double principal = 1000.00; // $1000 initial investment
        double interestRate = 0.035; // 3.5% interest rate
        int numOfYears = 7; // investment length is 7 years

        double finalValue = 0.0;
        finalValue = principal * Math.pow((1 + interestRate), numOfYears);

        System.out.println("Investing $" + principal
```

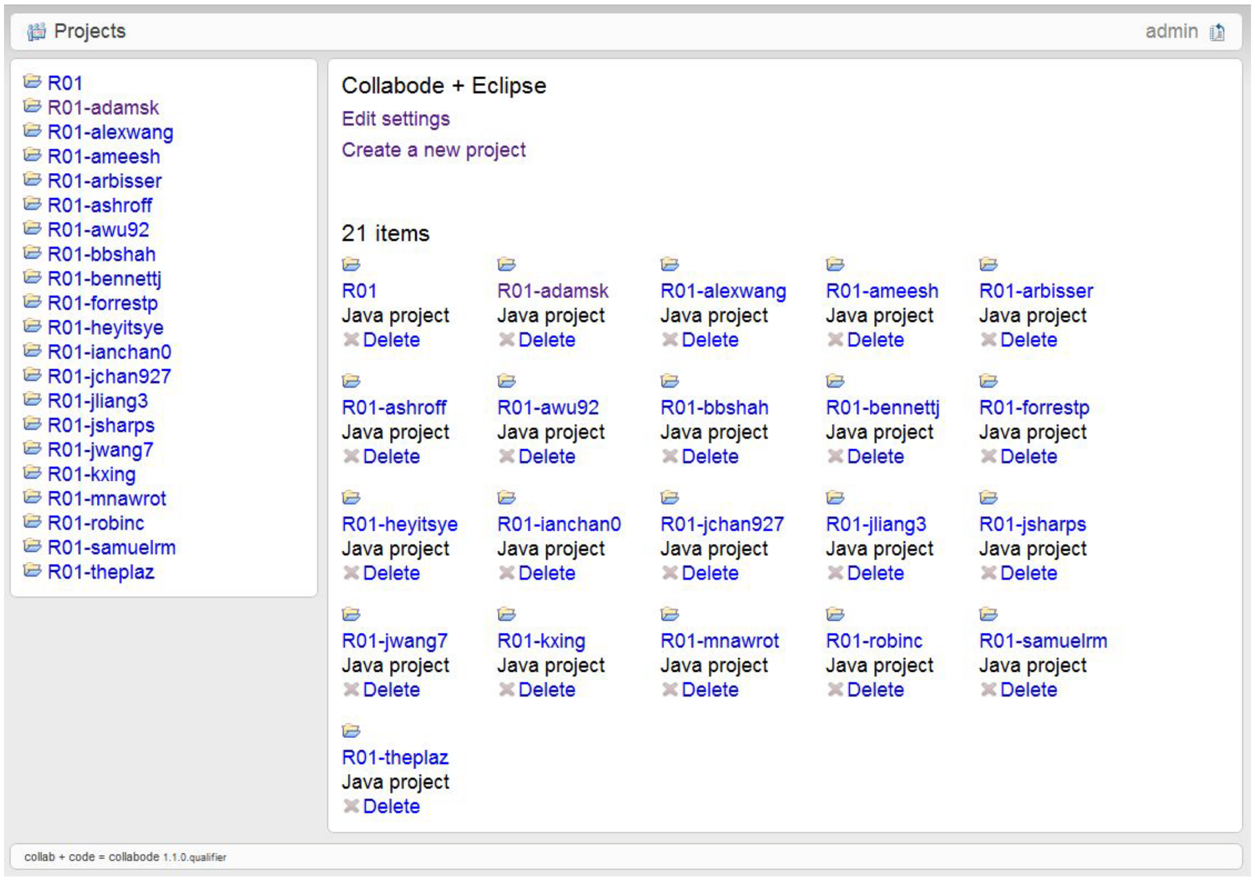
Figure 2-1. *A sample project open in Collabode.*



```
 [ Started Thu Sep 08 2011 12:33:21 GMT-0400 (EDT) ]
Investing $1000.0 at an interest rate of 3.5000000000000004% for 7 years will have a final worth of $1272.279262766573
 [ Stopped Thu Sep 08 2011 12:33:21 GMT-0400 (EDT) ]
```

Figure 2-2. *The Collabode console.*

Within the context of 6.005, Collabode was used in recitations as an easy way to distribute and observe student code. For each recitation, TAs created a single project containing starter code within Collabode, and each student was granted access to their own clone of the project. All of the clones were visible to the TA (Figure 2-3), so that any given student's work could be viewed in real time, and at the end of the recitation, no additional code collection needed to happen -- the TA's instance of Collabode already contained all student code in its workspace.


Projects

admin

R01
R01-adamsk
R01-alexwang
R01-ameesh
R01-arbisser
R01-ashroff
R01-awu92
R01-bbshah
R01-bennettj
R01-forrestp
R01-heyitsye
R01-ianchan0
R01-jchan927
R01-jliang3
R01-jsharps
R01-jwang7
R01-kxing
R01-mnawrot
R01-robinc
R01-samuelrm
R01-theplaz

Collabode + Eclipse
Edit settings
Create a new project

21 items

R01 Java project Delete	R01-adamsk Java project Delete	R01-alexwang Java project Delete	R01-ameesh Java project Delete	R01-arbisser Java project Delete
R01-ashroff Java project Delete	R01-awu92 Java project Delete	R01-bbshah Java project Delete	R01-bennettj Java project Delete	R01-forrestp Java project Delete
R01-heyitsye Java project Delete	R01-ianchan0 Java project Delete	R01-jchan927 Java project Delete	R01-jliang3 Java project Delete	R01-jsharps Java project Delete
R01-jwang7 Java project Delete	R01-kxing Java project Delete	R01-mnawrot Java project Delete	R01-robinc Java project Delete	R01-samuelrm Java project Delete
R01-theplaz Java project Delete				

collab + code = collabode 1.1.0.qualifier

Figure 2-3. *The instructor view of Collabode showing student clones of a project.*

From its deployment in 6.005 recitations, Collabode showed that it could support a sufficient volume of concurrent activity within projects hosted on a centralized server, and that it was a logical choice upon which to build the mobile instructor interface. The manner in which it was deployed in 6.005 recitations allowed us to easily intercept information passing through Collabode into the CollabodeTA interface.

2.4 Related Work

Previous work has been done in the areas of web-based IDEs, collaborative coding, and mobile visualizations. Amongst the numerous other active web-based IDEs projects are the Cloud9 IDE [8], built on the popular Ace editor [9] (recently merged with Mozilla's Skywriter project, formerly known as Bespin); the eXo Cloud IDE [10]; ShiftEdit [11], WaveMaker, a WYSIWYG Java Web 2.0 application development IDE [12], and WWWorkspace, a web-based Java IDE built on Eclipse [13]. These IDEs offer syntax highlighting for a variety of compiled and interpreted programming languages, code completion, project management, and integration with FTP and popular source control systems such as Git. Of these, only the eXo Cloud IDE supports real-time collaboration of up to 6 developers.

In the area of education, iTALC [5] has contributed a screen-sharing solution that allows instructors to have remote control over student screens in a number of ways. Its focus is less on collaborative learning between students and between instructor and student than on purely distribution of tasks from a top-down teacher-to-student direction, and the teacher using this system stands in front of a master computer with a view of each screen in the classroom. We see lapses in this design that we think this mobile lab instructor interface can fill. These include the ability to analyze (not just directly see) student progress, as well as freeing the lab instructor to move about the room using a mobile device rather than a fixed computer.

Technology-enabled active learning (TEAL) [6] places emphasis on "interactive learning", i.e. problem-solving in groups with active discussion, with shared visualizations of class concepts. Although TEAL is suited for learning concepts rather than producing or monitoring the production of code, the TEAL system incorporates means to gauge student understanding via periodic polling for and aggregation of student answers during the course of a typical class session.

Koile and Singer [7] have developed the Classroom Learning Partner (CLP) built on previous Classroom Presenter work, which was tested in introductory computer science classes. In this design, the lecturer annotates slides with digital ink and students likewise submit answers digitally. The instructor then gets histograms clustering student responses. Like the mobile instructor interface, CLP is tablet-based, from which we can draw some insight, although the core of the mobile instructor interface will be the Collabode IDE.

Chapter 3

CollabodeTA: A Mobile Instructor Interface for Collabode

3.1 Motivation

Our goal in creating CollabodeTA is to empower individuals not necessarily directly involved in the production of code towards a single project, but who monitor multiple implementations of the same project or assignment in parallel while maintaining the ability to interact with any of the implementations being overseen. In particular, we hope to address the student/instructor interaction and code collaboration problems presented in Sections 2.1 and 2.2 by using the technology Collabode provides in order to improve software engineering and programming education.

In a typical software development class at MIT, students are given programming assignments that can range from guided code templates to open-ended specifications. The distinctions between these project types are explained further in Section 3.2. These assignments may be given as individual projects, or occasionally, as group projects. During the time frame that students are given to complete their smaller-scale programming assignments or larger-scale projects, it is common to give students time to work in one or more of a software lab, recitation, or office hours.

Software labs

Software labs are times for students to have full access to any computing resources they might need, to serve as a fixed meeting time for group coding, or to serve as a fixed time during which students complete assignments that measure their understanding of recent lecture material. At MIT, software labs in which students are required to complete assignments and have their solutions checked off by instructors usually demand mandatory attendance. One or more instructors, teaching assistants (TAs), or lab assistants (LAs) are generally present in the lab to provide guidance and answer any questions that the students may have.

Recitations

Classes such as 6.005 use scheduled recitations as a time for TAs to review lecture material in smaller groups of students. A recitation session may also include time for the TA to demonstrate ideas by running bits of code in front of the class, or for students to complete small coding assignments individually or in groups under the TA's supervision.

Office hours

Office hours are designated times at which TAs or instructors make themselves available to answer student questions without presenting new material. They are distinguished from official software lab hours by being optional and held at the convenience of the instructor or students present. Unlike recitations, office hours are generally less structured, and instructors are not expected to formally present or review course material; however, students who attend are expected to bring questions for the instructor to answer.

What we have observed in all three of these meeting types -- software labs, recitations, and office hours -- is that students bring in their work on their own laptops or by connecting to their code on an external server via SSH. Students then work at their own pace individually or in small groups, waving instructors over or joining a shared help queue if they feel stuck, as described in Section 2.1.

This development process is near opaque to the instructors, who typically have little to no idea of what progress the students in the lab are making until they are explicitly called over to give feedback on their ideas or answer clarifying questions. Furthermore, instructors are usually seeing the students' code for the first time when help is needed, requiring them to spend time understanding the state of the students' code: how far they have progressed, what design decisions have been made, and what faults might exist in the code as written. Our final observation is that having a high degree of mobility is important to instructors in a lab setting. Because of the nature of having to observe many projects simultaneously, the more compact their teaching materials are, the more easily lab instructors can travel around a room to reach students in need of assistance.

Collabode gives us a platform within which all student code lives on a central server, and from which near up-to-the-second, keystroke-by-keystroke changes can be monitored. This creates the potential for a new type of interface for what we consider the "mobile instructor"; that is, an interface built to be used in conjunction with Collabode to monitor student progress in a lab setting, with an emphasis on visualizing student progress, aggregating classroom performance metrics, and portability for the instructor.

A mobile instructor interface would provide means for lab instructors to accomplish the following critical tasks, all of which we have seen are currently infeasible and/or difficult to accomplish with current software lab technologies, and which contribute to a holistic view of the classroom:

- Monitor student progress and rate of completion of assignments
- Identify outliers within the students or student groups that are having notable successes or difficulties with the given assignment
- Identify common mistakes made by students due to misunderstandings of the course material

CollabodeTA is the mobile instructor interface we have designed to fill this gap in software engineering and programming education. By making these critical tasks possible, we hope that instructors will more accurately be able to gauge progress and understanding early in the teaching progress and proactively identify students who both need help and can help their peers during the course of one teaching session.

3.2 Types of Software Lab Assignments

Software engineering has many configurations and degrees of flexibility in design. For the purposes of this discussion, we will find it helpful to break down software projects into three different categories.

Structured

Structured assignments are built around a code skeleton or template, with specific methods or areas where students are instructed to work. This is the most straightforward type of project for which to monitor progress, as student progress can be measured in discrete and consistent chunks corresponding to each section of the code they are instructed to complete.

In most cases, the correctness of individual methods can be tested by using unit tests corresponding to each task, designed such that students incrementally pass these tests as portions of the template are completed. A formal unit test framework such as JUnit may be used, or the results of unit tests may simply be printed to standard output in the user's console. In other cases where tests are not provided but the assignment skeleton is well-established or the scope of the assignment is small, student progress can still be gauged based on how much of the skeleton is completed, and how much code the student has contributed to each section compared to how much code is expected from the student to produce a working solution.

An example of a structured assignment is as follows:

```
Complete the implementation of the isValidMove(Coordinate) method for
each type of playing piece in a chess game.  This method should return
true if the given coordinate is a valid move for that piece based on
its current location and the rules of the game, or false otherwise.
```

Interface

In interface-based assignments, students are given a specification or a common interface with some design freedom as to how to implement that interface. In this type of project, the expectations are expressed in the project specification, but the structure of the implementation is up to the student. A suite of unit test cases testing completion of the specifications, where student progress is monitored by the rate at which their designs fulfill specifications, can be used to gauge progress on such projects; however, the amount of code written, whether on an absolute scale or relative to the rest of the class, while one factor on the step to completion, is not a reliable indicator of student progress in these types of assignments.

Here is an example of an interface-based assignment using the same chess game example as before, assuming the chess game contains an AI component where a human player can play against the computer. Note that the assignment does not specify an algorithm or restrict which intermediary data structures the student uses, and provides only the specification for the method:

```
Implement the computeNextMove() method for the computer player in the
chess game, which returns the playing piece and coordinates of the
player’s next move.
```

Freeform

Freeform assignments are those in which the students are given a set of requirements and otherwise have freedom to design and implement them how they wish, while being handed minimal starter code. Intermediary assignments for open projects may be more structured; for example, students may be requested to submit design documents. This is the most difficult type of project for which to monitor progress, but one can imagine students being required to write their own test cases, and submitting those as intermediary assignments, which structures freeform assignments towards a more interface-based structure.

An example of a freeform assignment is as follows:

```
Implement a chess game, where users can play against each other or the
computer.  You must accurately respect the rules of chess, but are
otherwise free to implement the game board, pieces, user interactions,
etc. however you wish.
```

3.3 Metrics

Collabode records real-time, keystroke-by-keystroke data as students are working in class. As a result, there are a number of ways in which we could have decided to achieve our goals of measuring student performance as concisely and accurately as possible. It was important to us to choose metrics that would be meaningful when viewed over time, and which would be meaningful when aggregated, and viewing the class as a whole. Our experience observing and interviewing the instructors of the classes mentioned in Section 2.1 helped us to narrow the list of interesting metrics down to the following categories, shown with some sample questions that could be answered with each category of metrics:

- **Unit test results:** Given a test suite for the assignment, how many unit tests has the student's current implementation passed?
- C**ompiler metrics:** How many errors and/or warnings does the compiler find in the student's code?
- **Run-time metrics:** What exceptions are thrown when the student runs their code? What non-fatal errors does the student encounter?
- **Student-produced metrics:** How much code has the student written? How many lines of code were written compared to the amount of code deleted? If there are multiple parts to the given assignment, which assignments have been completed

and which are the student still working on? How long did it take for the student to complete each task?

- **Integration metrics:** The number of *integrations* made by Collabode of a student's code indicate how many complete chunks of compile-error-free code the student has written. How many of these have there been?

Each of these questions also become more meaningful when we ask "how do these numbers change over time? How does the student's performance now compare to how they were doing 10 minutes ago?" Similarly, we could also ask "what is the majority of the class scoring for each metric? How does this student compare with the class mean, minimum, or maximum?"

For CollabodeTA, we began the process of measuring student performance by analyzing student console outputs. This made sense for a variety of reasons. First of all, looking at console outputs allows us to measure student performance via multiple categories of metrics, specifically those collected at run-time and those which were student-produced. Printing output to a console is often a critical part of the development and debugging process, and looking at such output is often essential for identifying and debugging problems in code, and in discerning what students were thinking or trying when errors were made. 6.005 recitations in the Fall 2011 semester used assignments that primarily relied on either achieving a correct value or (when tests were provided) testing via Java main methods that printed expected output to the console when correct. Therefore, the structure of our data also supported analyzing console output over starting with other metrics such as unit test cases, which were not available at the time. We found it convenient that the collections of discrete, smaller-scale tasks assigned in 6.005

recitations tended to fall under the structured and interface-based assignment types, and were therefore naturally testable via console output because of the ease with which individual tasks could be tested separately. In all, we believe that the work done with student console output is useful for the broadest variety of assignments, and in particular, was most the most meaningful approach for the data with which we had to evaluate our system.

3.4 Design Overview

Our design of the mobile instructor interface is to intrude as minimally as possible on the lab instructors' attention and natural workflow while still augmenting their understanding of what is happening in the classroom at any given time. We therefore aimed to summarize and present data in the interface such that can be understood "at-a-glance" with minimal user interaction. This section provides a general overview of components and terminology that we will use to describe the Collabode mobile instructor interface. Following sections will describe in more detail the motivations, design, and implementation of each feature of the interface.

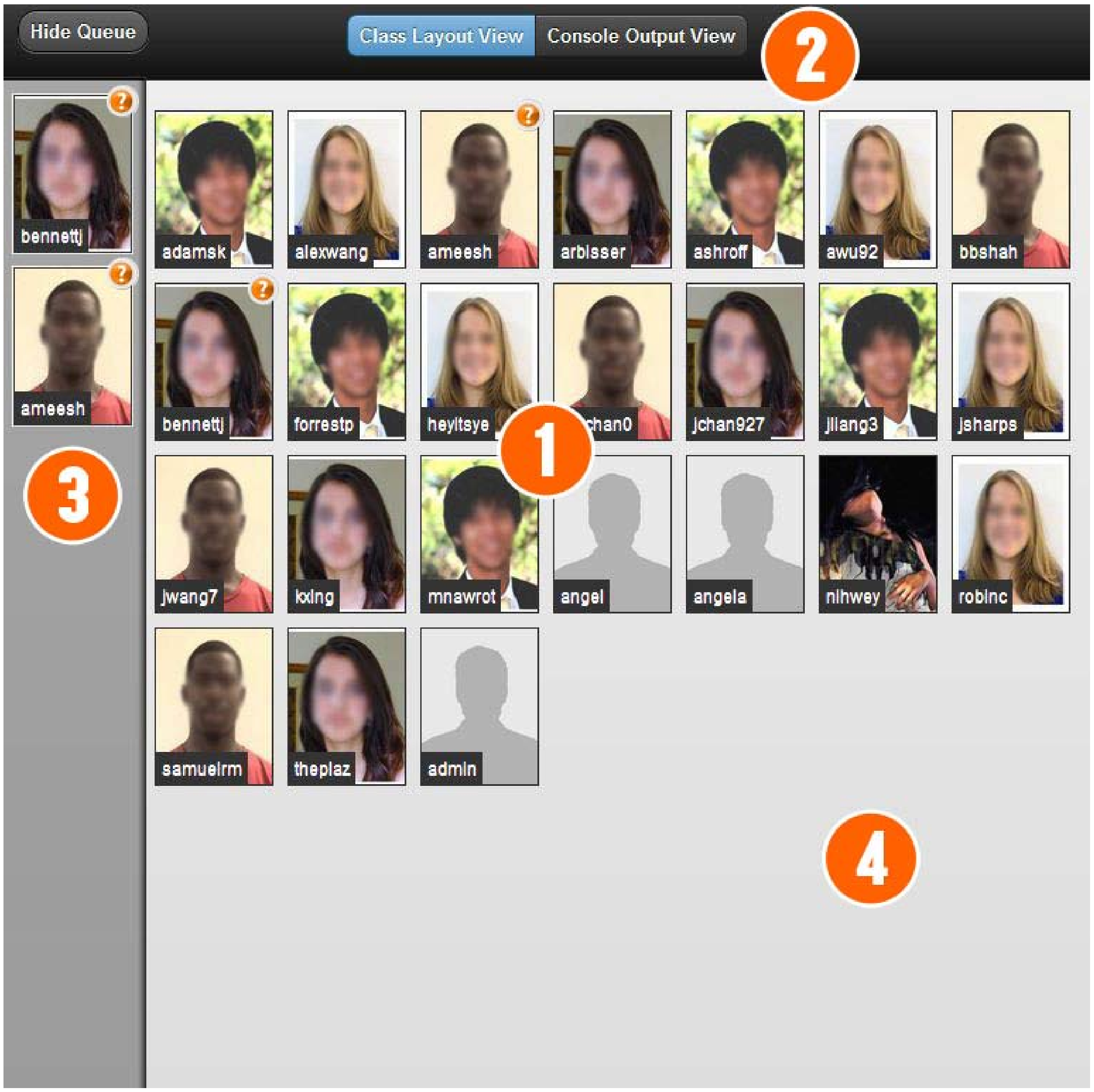


Figure 3-1. *Overview of CollabodeTA mobile instructor interface components. 1) Student cards, 2) Toolbar, 3) Queue tray, 4) Tray for alternate views (i.e. class layout, console output, student panels). Class layout view is shown.*

3.4.1 Student Cards

A student card is designed to concisely summarize information about a student. It consists of their picture, which can be taken from the 6.005 class website (or a placeholder image if a picture does not exist), a small name tag with their Collabode username, and any additional decorations (which we will refer to as *badges*). The only badge currently used is an orange ? icon indicating that a student is on the help queue, but future work may incorporate badges corresponding to metrics such as test suite completion, amount of code output, and help status on the queue.

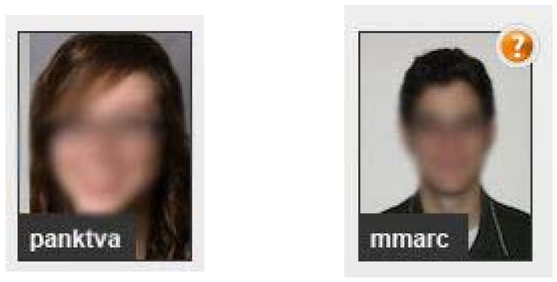


Figure 3-2. *A plain student card (left). A student card decorated with a ? badge (right).*

Student cards were designed to be viewed in aggregate in the class layout view (Section 3.5), in the help queue tray (Section 3.8), and as a means to access student panels (Section 3.7).

3.4.2 Toolbar

The toolbar is always visible and provides a way to switch between the main views of our interface. It also allows toggling of the help queue visibility. The button corresponding to the current view is highlighted in the toolbar.

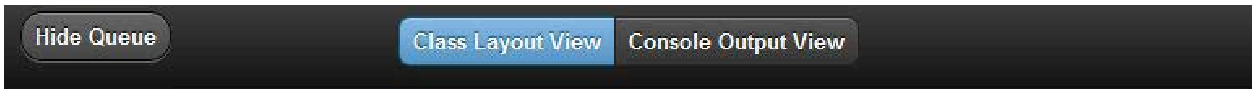


Figure 3-3. *Close-up of the toolbar.*

3.5 Class Layout View

3.5.1 Motivation

Our original motivation for the class layout view was to provide a concise, at-a-glance summary view of the entire class's activity and aggregate performance statistics. However, from our interviews of people who have previously led 6.005 labs and labs in other programming or software engineering courses, we discovered that a common (although tangential) problem experienced by lab instructors was that of locating students in labs given just their names provided from a help queue. This problem is especially apparent in the first few weeks of class (or simply large classes in general) when instructors have not yet had a chance to familiarize themselves with student faces. One current solution to this problem is to call out student names and ask them to raise their hands, which is time-consuming to do in large classes in addition to being distracting to other students. Another solution currently used in the 6.005 queue is to have students self-identify their location when they join the class help queue. Rather than creating a second view to solve this student recognition problem, we decided that the class summary view could be augmented to solve this problem as well, which resulted in this final list of design goals:

1. Summarize class activity in a display that can be understood at a glance and requires no human interaction

2. Incorporate student photographs to help instructors learn and recognize faces and Collabode usernames
3. Incorporate information about the students' locations in the classroom, to help mobile instructors locate students they are trying to help

The class layout view (Figure 3-4) displays students in the class represented by their student cards, and allows the user to drag and drop the cards into a spatial arrangement (Figure 3-5) corresponding to students' physical location in the room. Filters accessed by the student panels (described in Section 3.7) fade and highlight students that match different criteria. In combination with highlighting and badges on student cards, we believe that we have thus achieved our design motivations in a space-efficient way.

The largest unsolved aspect of this design is the generation of the spatial arrangement. This class layout generation turns out to be time consuming to do manually in practice, especially when student locations are not assigned and therefore change from class to class as they are given freedom to work in their own groups. Furthermore, student locations cannot be quickly or easily determined by the instructor until after the majority of students have arrived, which is also a problem.

We did not prioritize the class layout generation in this thesis, but the chapter on future work (Chapter 5) includes a discussion on ideas for arranging student cards in the class layout view with minimal human effort.

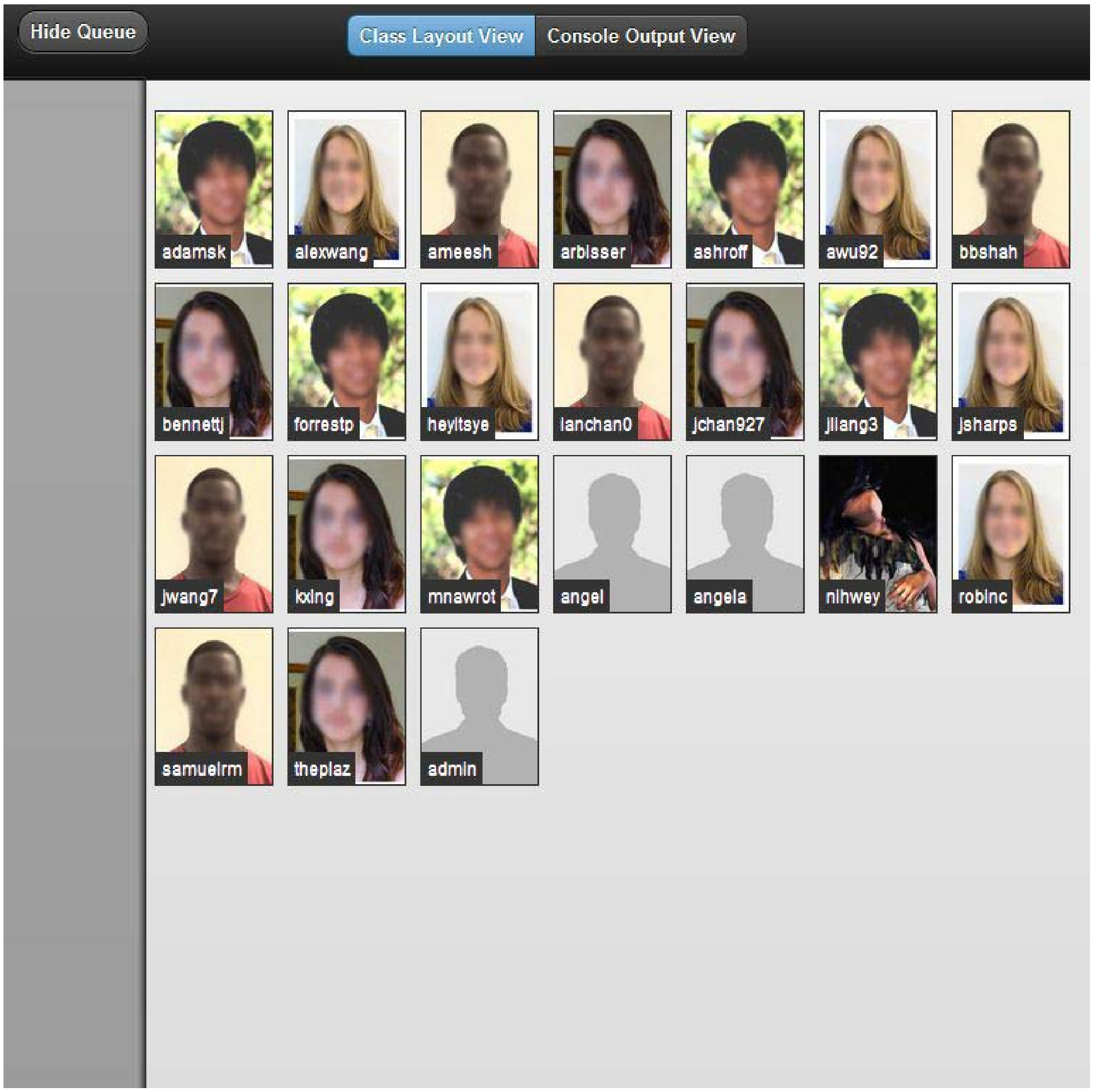


Figure 3-4. *Class layout view with student cards arranged in a grid.*

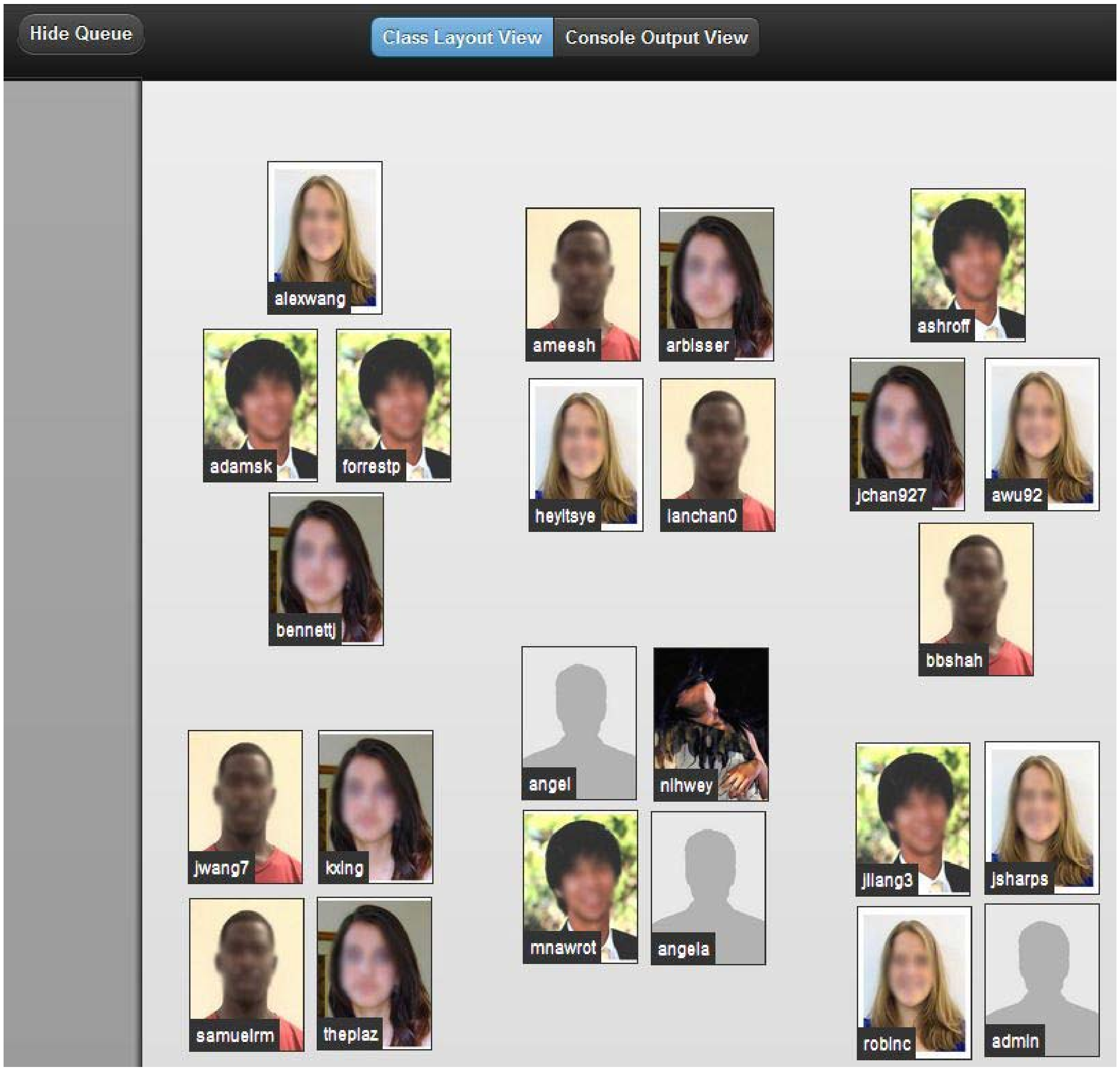


Figure 3-5. *Class layout view with student cards arranged in groups of four.*

3.5.2 Implementation

The implementation of the class layout view is straightforward. The view itself fills the right tray when selected, and student cards are jQuery UI [15] draggable objects within the droppable view space. Student card sizes are optimized such that a class of about 30 students can fit comfortably in an arrangement on an iPad screen. Logic to optimize student card sizes for significantly larger class sizes was out of the scope of this thesis, but is an interesting problem and mentioned in the section on future work (Section 5.1).

3.6 Console Output View

3.6.1 Motivation

Section 3.3 described why we are interested in examining the output students get in the console while working on a variety of assignment types as a measure of demonstrated student understanding. Console outputs can be used for more than simply debugging or identifying the source of one students' bug; especially when compared to expected output from a test method, they can be used as a measure of overall class understanding, and as a result help an instructor adjust the pace or content of a recitation-like class or lab session during the course of the lab to more effectively address students' most commonly misunderstood lesson topics. We therefore designed this view to allow lab instructors to see the outputs of every current student simultaneously, and like the class layout view, require minimal-to-no human interaction so as to be as glanceable as possible.

3.6.2 Types of Console Output

While users may print any arbitrary text to their program console, console output from recitation data we used to evaluate CollabodeTA fit into a few major categories.

Test function output

Comparing actual console output to expected output of test functions is, assuming academic honesty, the most immediate way to tell whether a student has correctly and completely implemented the function being tested. Examples of test function output from 6.005 recitation data included explicit correct/incorrect print statements (A), or print statements containing values or print statements of data structures that were expected to have been computed correctly, (B, C).

A.
```
1)CORRECT:  array1 and array2 contain the same elements!
2)CORRECT:  list1 and list2 contain the same elements!
3)CORRECT:  string1 and string2 contain the same values!
4)CORRECT:  stringNull1 and stringNull2 contain the same values!
```

B.
```
Investing $1000.00 at an interest rate of 3.5% for 7 years will have a
final worth of $1272.28.
```

C.
```
[King of Diamonds, King of Spades, Ace of Diamonds, Queen of Diamonds]
```

Nondeterministic output

A separate class of console outputs arises when, as in some 6.005 recitations, the assigned task deals with randomness (A), timing (B), synchronization (C), or otherwise results in output that is nondeterministic and varies from run to run and student to student.

A.
```
Results of 10 dice rolls: {1:2, 2:1, 3:1, 4:2, 5:2, 6:2}
```

B.
```
Timing method findPrimes: 461 milliseconds
```

C.
```
thread1                vs.                thread2
thread2                                   thread3
thread3                                   thread1
```

Exceptions/Errors

Unexpected output occurs when students run into exceptions or errors in their code. We observed that a standard Java stacktrace was usually printed when exceptions occurred, and other uncaught errors such as infinitely looping or recursive functions that printed output indefinitely were characteristic and easy to spot.

Arbitrary

The catchall category for console output includes all other print statements which students include for the sake of debugging or even sometimes is just due to typos.

In general, console output from test functions written by the course staff that have predictably correct or expected text are the types of outputs that this CollabodeTA view focuses on. Assuming that correct output from student code will match that of the staff implementation allows us to also assume that any differences from the “correct” output that are not apparently due to nondeterminism or arbitrary debugging are due to erroneous code. Once these cases are identified, instructors can then proceed to group error cases into larger assignment-specific categories.

3.6.3 Implementation

For the purposes of this thesis, which was evaluated on data from a large number of pre-recorded recitations, it was important to devise both an efficient design for clustering and rendering student outputs through time as well as an efficient database schema to support storing all of this computed metadata. To establish some terminology:

- At any given point in time, we consider an *output* to be the textual content of the student's output console, which is fed from both standard output and standard error streams.
- A *canonicalization* (or *normalization*) of an output is the transformation of an output to a form that can be compared to other outputs accurately. For example, in our implementation, whitespace is trimmed from the beginning and end of each output during the canonicalization process. A discussion of optimizations to the canonicalization process is continued in section 5.1.4.
- A *cluster* of outputs is a set of outputs whose text maps to the same canonical value.

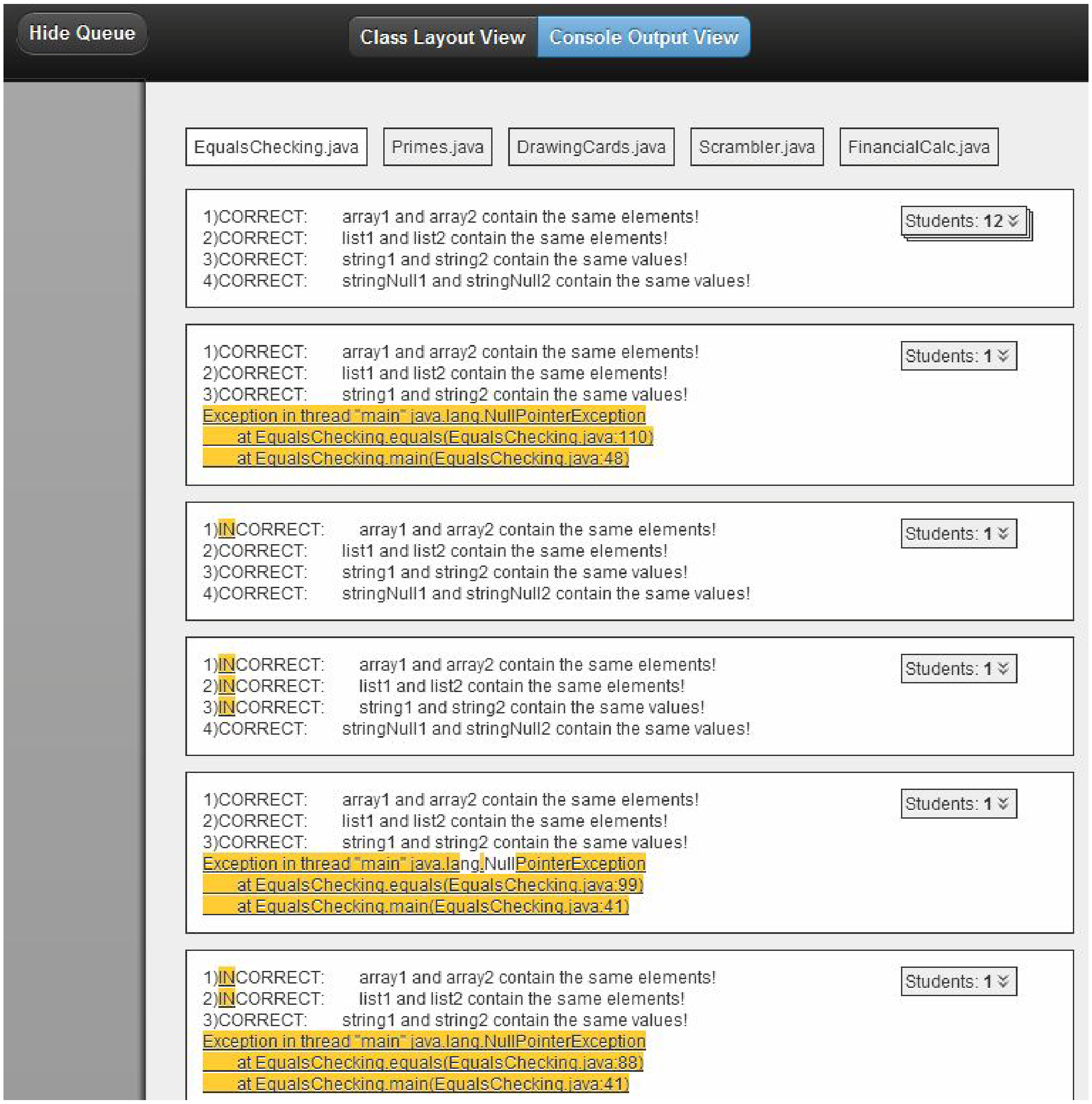
All raw outputs are normalized to their canonical form before clustering begins, and CollabodeTA shows only normalized clusters. Section 3.9.3 contains details about optimizations made in our clustering algorithm implementation.

Figure 3-6 shows a sample console output view. Our design displays all the variations of student outputs (i.e. all the clusters) at a chosen point in time, sorted in order of decreasing occurrence. We found the cluster with the highest number of occurrences interesting because it indicates what most of the students in the class are encountering at any given time. From replaying our recorded data, we found that there was often a clear plurality or small set of frequently-occurring clusters at any given time, and that students tended to converge on the correct output cluster by the end of the time allotted to work on the given assignment.

At a given time, we consider the cluster with a plurality of occurrences (or an arbitrarily chosen one from the set of maximally occurring clusters, if there is no single cluster that occurs most frequently) to be the *prototype* cluster. The other clusters are displayed below the prototype cluster in decreasing order of occurrence. Our solution for making this glanceable was to perform two-way textual diffs of each cluster with the prototype cluster and to display subsequent (non-prototype) clusters with the diffs highlighted. Insertions are underlined, while deletions are displayed with a strikethrough. We used the Google diff-match-patch Javascript library [21], which implements the Myer's diff algorithm as well as provides some optimizations for semantically cleaning up diff results.

We also made some rendering optimizations to the displayed diffs. Because the result of a diff is a sequence of insertion, deletion, and matched segments, we quickly realized that it was not necessary to show every one of these segments. For example, a modification of a chunk of text naively appears as a deletion immediately followed by an insertion. Similarly, blank output appears as a deletion of the entire prototype text, but this is more clearly shown (and more space-efficient) simply as blank output. We therefore only display deletions if they are not followed directly by an insertion. Figure 3-7 shows the difference between our diff rendering compared to a naively rendered diff.

Finally, we encountered some cases in our replayed data of long output (e.g. long test output, output from infinite loops), so we set a maximum height for each console output and require that the user scroll the inner frame holding the cluster text to see more of a particularly long output. This decision was made to save significant screen space, especially in the case of the stack overflow problem.



Students: 12

```
1)CORRECT:      array1 and array2 contain the same elements!
2)CORRECT:      list1 and list2 contain the same elements!
3)CORRECT:      string1 and string2 contain the same values!
4)CORRECT:      stringNull1 and stringNull2 contain the same values!
```

Students: 1

```
1)CORRECT:      array1 and array2 contain the same elements!
2)CORRECT:      list1 and list2 contain the same elements!
3)CORRECT:      string1 and string2 contain the same values!
Exception in thread "main" java.lang.NullPointerException
      at EqualsChecking.equals(EqualsChecking.java:110)
      at EqualsChecking.main(EqualsChecking.java:48)
```

Students: 1

```
1)INCORRECT:      array1 and array2 contain the same elements!
2)CORRECT:      list1 and list2 contain the same elements!
3)CORRECT:      string1 and string2 contain the same values!
4)CORRECT:      stringNull1 and stringNull2 contain the same values!
```

Students: 1

```
1)INCORRECT:      array1 and array2 contain the same elements!
2)INCORRECT:      list1 and list2 contain the same elements!
3)INCORRECT:      string1 and string2 contain the same values!
4)CORRECT:      stringNull1 and stringNull2 contain the same values!
```

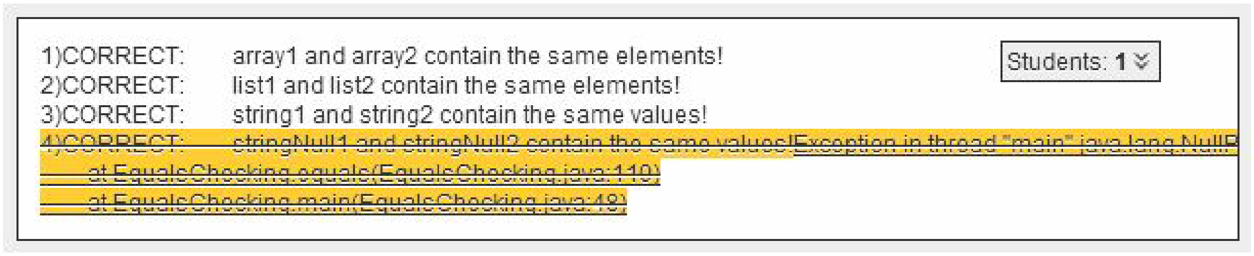
Students: 1

```
1)CORRECT:      array1 and array2 contain the same elements!
2)CORRECT:      list1 and list2 contain the same elements!
3)CORRECT:      string1 and string2 contain the same values!
Exception in thread "main" java.lang.NullPointerException
      at EqualsChecking.equals(EqualsChecking.java:99)
      at EqualsChecking.main(EqualsChecking.java:41)
```

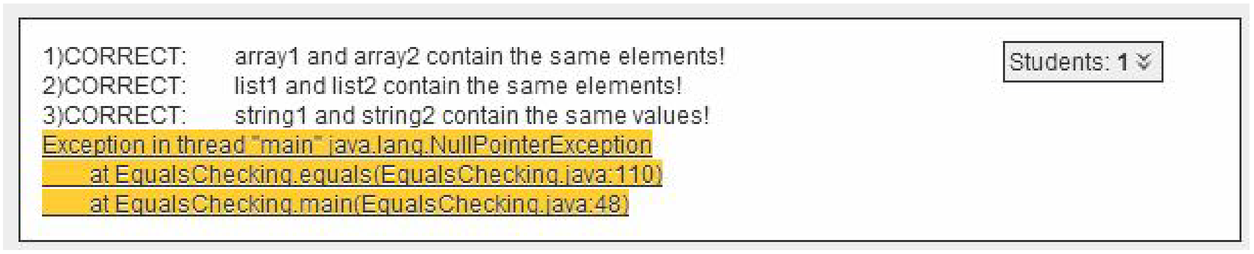
Students: 1

```
1)INCORRECT:      array1 and array2 contain the same elements!
2)INCORRECT:      list1 and list2 contain the same elements!
3)CORRECT:      string1 and string2 contain the same values!
Exception in thread "main" java.lang.NullPointerException
      at EqualsChecking.equals(EqualsChecking.java:88)
      at EqualsChecking.main(EqualsChecking.java:41)
```

Figure 3-6. *Console output view.*



```
1)CORRECT:     array1 and array2 contain the same elements!
2)CORRECT:     list1 and list2 contain the same elements!
3)CORRECT:     string1 and string2 contain the same values!
~~4)CORRECT:     stringNull1 and stringNull2 contain the same values!Exception in thread "main" java.lang.NullP~~
~~        at EqualsChecking.equals(EqualsChecking.java:110)~~
~~        at EqualsChecking.main(EqualsChecking.java:48)~~
```



```
1)CORRECT:     array1 and array2 contain the same elements!
2)CORRECT:     list1 and list2 contain the same elements!
3)CORRECT:     string1 and string2 contain the same values!
Exception in thread "main" java.lang.NullPointerException
        at EqualsChecking.equals(EqualsChecking.java:110)
        at EqualsChecking.main(EqualsChecking.java:48)
```

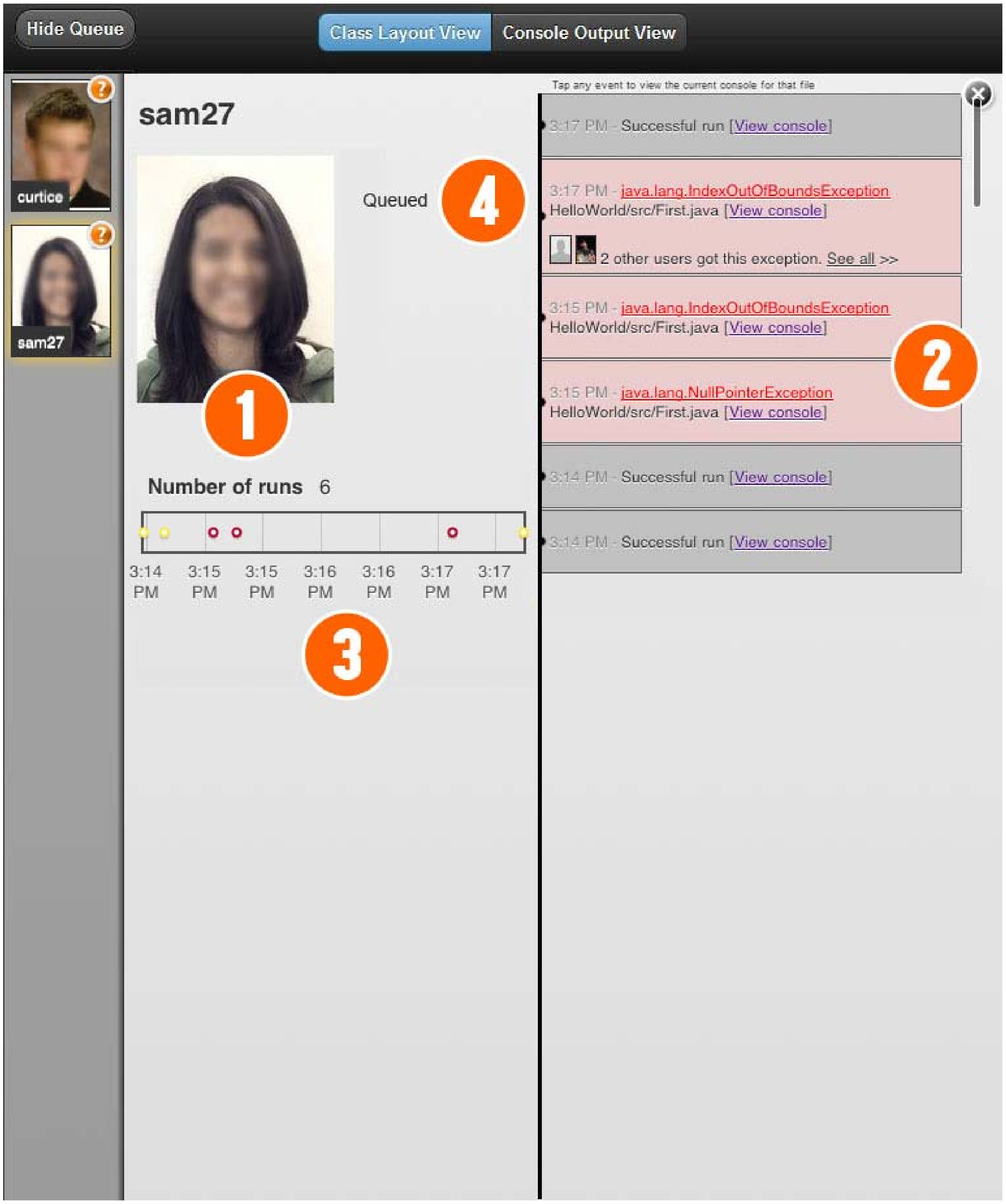
Figure 3-7. *A naive diff (top) compared to a cleaned diff (bottom).*

User studies (Chapter 4) revealed that users would find it useful to scroll back in time to snapshots of individual student performance (or the performance of the entire class) even during the course of the recitation session. Because of the way we have designed the database schema and cluster computation, this can be done with minimal effort. Section 5.1 suggests designs and ways to improve performance such that this functionality can be included in our interface for use in real time.

3.7 Student Panel View

3.7.1 Motivation

The *student panel view* summarizes an individual student's activity during a lab session. The student panel is a way for instructors to see more detailed information about a specific student, as well as link to the student's code, actual console, and other documents.



Tap any event to view the current console for that file

3:17 PM - Successful run [View console]

3:17 PM - java.lang.IndexOutOfBoundsException
HelloWorld/src/First.java [View console]
2 other users got this exception. See all >>

3:15 PM - java.lang.IndexOutOfBoundsException
HelloWorld/src/First.java [View console]

3:15 PM - java.lang.NullPointerException
HelloWorld/src/First.java [View console]

3:14 PM - Successful run [View console]

3:14 PM - Successful run [View console]

Figure 3-8. *A student panel. 1) Student information, 2) Event stream, 3) Timeline, 4) Queue status widget*

3.7.2 Implementation

In our implementation, tapping on a student card displays that student's panel in the right tray. Our design shows the following information:

- Basic information about the student (their picture and Collabode username)
- An *event stream* of the selected student's activity. This event stream logs activities such as successful executions of source files and unsuccessful executions with the Java exception that occurred. Error events link to other students who have gotten the same error, which we found is helpful for determining error frequencies. Each event in the event stream links to the appropriate file console or file for that activity. If users with similar exceptions in the event stream appear, they can also be clicked to filter the class layout view, as shown in Figure 3-9.
- A simple timeline of student activities over time, with tappable markers linking to events in the event stream, which can grow to be numerous during the course of an hour-long lab session.
- A widget indicating the student's help queue status. From the student panel, instructors can see whether the selected student is currently on the help queue (improving glanceability in the case of long queues), and mark them as being helped or remove them from the queue after being helped. Other information that could easily be included is the latest time a student was helped, how many times they have been helped, and other statistics of interest to instructors. Section 3.8. describes the help queue in more detail.

As CollabodeTA is expanded to record other metrics, similar detailed information relevant to individual students should be included in the student panels for easy access.

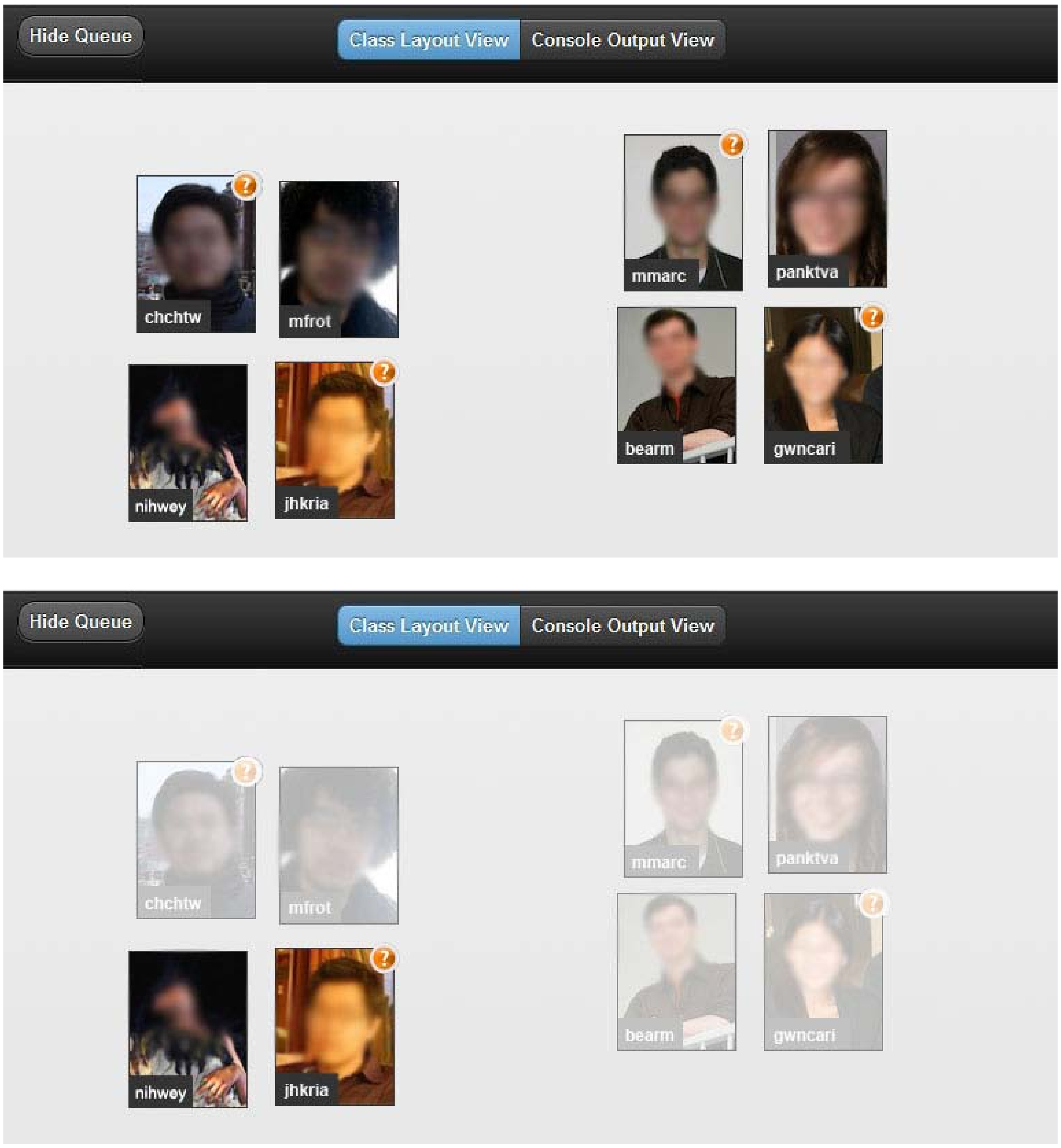


Figure 3-9. *Class layout view before and after filtering from the event stream.*

3.8 Help Queue

3.8.1 Motivation

A number of MIT software classes use a formal help queue in labs to line students up for instructor help in a reasonably fair (first in, first out) way. The 6.005 Karga queue [17] and 6.004 (Computation Structures) help and lab checkoff systems [16] are two examples of software lab help queues.

The mobile instructor interface also includes a simple implementation of a help queue. Unlike the other mobile instructor features mentioned so far, our hope is that the inclusion of a help queue will 1) increase fairness by allowing students to elect for help even if they do not stand out to instructors by other metrics, and 2) provide a way for multiple instructors in the same session to distribute student question load given the additional information that our interface provides.

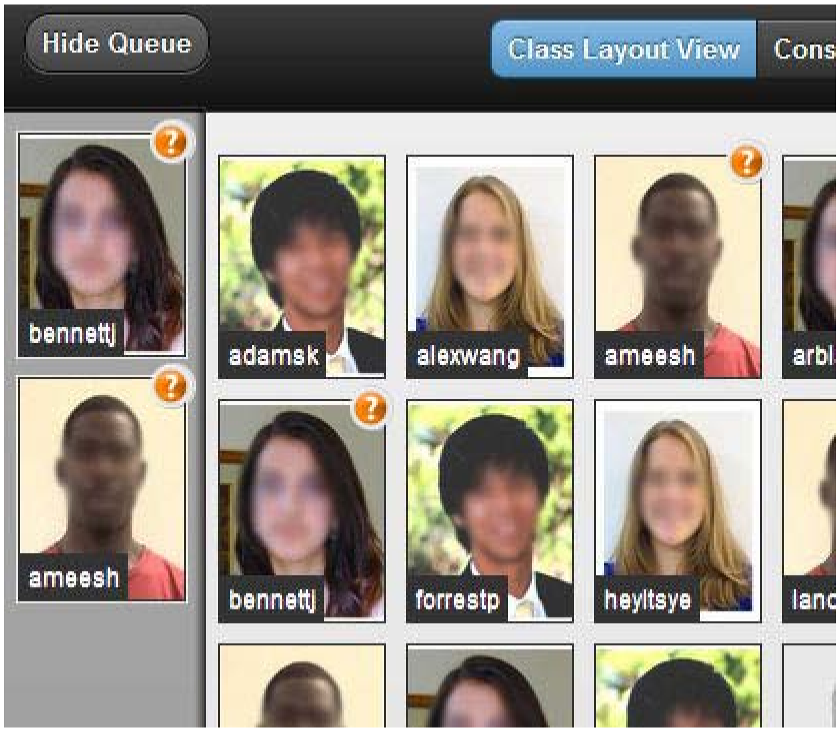


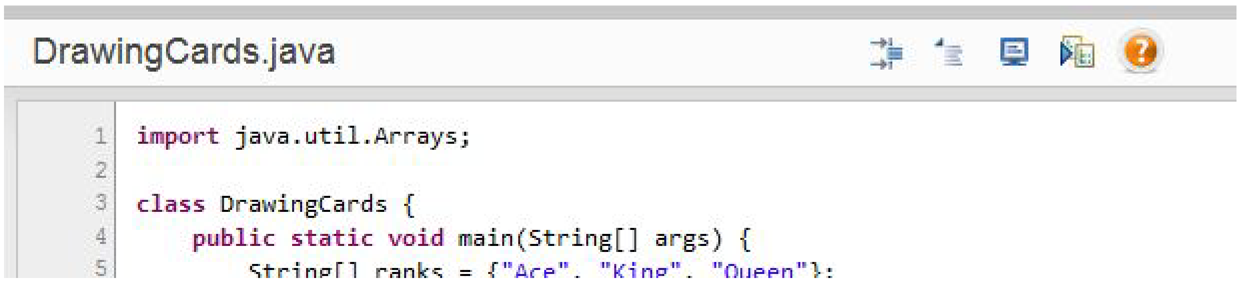
Figure 3-10. *Close-up of the help queue and highlighted students in the class layout view.*

There is currently no implemented student UI for the help queue. However, a student can only appear once on the help queue, and their earliest position on the queue is maintained over any later joins while they are still on the queue. There is no limit to how many times students may join after being removed from the queue.

3.8.2 Implementation

The mobile instructor help queue is displayed as a panel to the left of the interface. It fills with student cards from top to bottom as students join the queue. Queue visibility can be toggled by clicking the Show Queue/Hide Queue button on the left of the main toolbar (directly above the queue itself), and can therefore be either constantly visible or hidden when the queue is not active to provide more space on the right for content. The queue does not ever cover other content, and when the queue panel is hidden, whatever content is currently displaying on the right expands to fill the entire screen. It is important that the queue panel behave this way, as the width of the queue is strictly larger than a single student card, and therefore student cards can be hidden by the queue panel when in the class layout view.

A small ? badge marks students in the class layout view who are on the queue so that they are immediately apparent to the instructor, who might decide to target a group of students based on their location and queue status. Students elect to join the queue by pressing the same ? icon in their individual editors (as shown in Figure 3-11) and appear immediately in all instructor interfaces, if there are more than one.



```
import java.util.Arrays;

class DrawingCards {
    public static void main(String[] args) {
        String[] ranks = {"Ace", "King", "Queen"};
```

Figure 3-11. *Help queue button integration with the student toolbar.*

The student panel view also displays a student's queue status. Tapping on a queued student's card brings up the student's panel in the same way that tapping on the card in the class layout view does. A student's queue status cycles between three states:

1. NOT_QUEUED: the student is not currently on the queue
2. QUEUED: the student is on the queue but not being helped by an instructor
3. HELPED: the student is being helped by an instructor

By choosing to make assign one of three states to any given student, instructors can see which students are already being helped by other instructors in the class without having to remove the student from the help queue. A student card is not removed from the queue until an instructor marks them as HELPED and subsequently removes them from the queue.

3.9 Replaying Recitations

This section describes the components of Etherpad, schema design, methods, and optimizations that were relevant to replaying recorded 6.005 recitations so that real-time use of CollabodeTA in a classroom could be simulated.

3.9.1 Structure of a Replay

The recitation data gathered from Collabode comes in two parts. The first is a complete workspace directory containing the Collabode plugin, all other plugins and metadata necessary for Collabode to run properly, and a complete copy of the master project and every student's cloned project in its final state (i.e. as it was at the end of the recitation). Each project is a complete Eclipse Java project. Instructors may have their own project clone as well. The second half of the recitation data consists of the Etherpad database files. The essential files in this data are a properties file describing the database type, a log file of all committed database transactions executed since the Collabode server was started, and a script file that stores database information so that it can be recovered between restarts of the Collabode server.

In all, Figure 3-12 shows the high-level hierarchy of replay files:

```
workspace/
        .metadata/
        rec1/
        rec1-student1/
        rec1-student2/
        …
db/
        edits.log
        edits.properties
        edits.script
```

Figure 3-12. *Organization of files included in recitation data*

An Etherpad document is called a *pad*, and user edits are packaged into *changesets* which are applied to pads to update their content. In Collabode, both source files and user consoles are pads, but with different properties. Each pad has a unique ID from which we can determine its pad type and author, amongst other attributes.

3.9.2 Playback

Replays are captured in three phases.

In the first phase, we use the Etherpad API to read the database log and retrieve every revision in the database, where a *revision* is an object with attributes such as author, source pad ID, timestamp, and changeset. The revision attributes of interest to us are stored into a separate replay database table.

In the second phase, we run through each of the revisions we have collected and execute them in Collabode as if they were happening in real time. During this phase, user edits to files are reapplied as if the system were running live, which changes the state of the system as if it were running live and allows us to capture metrics that were not captured the first time, such as information about console output and exceptions that were being thrown. These metrics are again stored into the database so that this phase, which by involving the rest of the Collabode system is the most expensive phase to execute, need only be executed once per database schema change. During this phase, we assume all source files have an associated pad and check also to see if they have an associated console pad as well by attempting to access the console pad with the corresponding pad ID to the source file pad. If they do, we also collect the console output at each revision.

The final phase of playback is the rendering phase, where we assume we have collected all data of interest and render it to the CollabodeTA web interface. Except for the very first time a replay is run, this is the only phase that the system will need to go through, which significantly improves the speed at which replays can be played back.

3.9.3 Cluster Optimizations

During initial playback, all raw outputs are normalized to their canonical form before clustering begins. Post-canonicalization, clusters are identified by simple string comparison. Therefore, cluster storage in the database can be optimized by mapping each cluster -- which corresponds to a unique canonicalized string output -- to a unique id. Revision data stored in the replay database therefore need only store a cluster id, and the complete output text can be retrieved by looking up the text associated with the revision data stored in the replay database. This reduces the amount of storage for replay data significantly.

To significantly optimize cluster visualization, cluster ids for each student's console output after each revision are computed and stored upon running the replay of a particular recitation session for the first time. Thus, on subsequent replays of the same recitation data, each student's most recent precomputed cluster ids (optionally, the most recent precomputed cluster ids for each separate assignment's output console) can be retrieved at any desired point in time and we need only render the resulting set of clusters in our interface without having to renormalize and reassign clusters every time.

3.10 CollabodeTA Implementation

The CollabodeTA is directly integrated into the Collabode infrastructure. It is accessible via `http://<root_url>:<port>/mobile` and is protected by the same access control logic as the root Collabode project page.

The bulk of the CollabodeTA mobile instructor interface application logic is written in client and server-side Javascript with some Java. Embedded JS templates and Less JS are used for templating and styling. Notably, its design and touch behaviors are currently optimized for the Apple iPad 2 locked in portrait orientation. This orientation was chosen in order to show output cluster diffs in the console output view most efficiently. Like Collabode, the CollabodeTA mobile instructor interface can run on any Javascript-enabled mobile browser with a network connection to the Collabode server.

Chapter 4

Evaluation

We evaluated CollabodeTA in two ways. The first, described in Section 4.1, was to evaluate the usefulness of the mobile instructor interface by asking software lab instructors to use the interface in a simulated real-time lab setting. The second, described in Section 4.2, was to show the usefulness of CollabodeTA in post-lab analysis; that is, as a tool for analyzing how effective lab exercises were *after* the lab session was completed, without the time or attention constraints imposed by a real time lab or recitation imposes, for the purposes of evaluating the lab exercises themselves and for planning future classes.

4. 1 Mobile Instructor Interface User Studies

Our evaluation of CollabodeTA was performed using pre-recorded data collected from the Fall 2011 semester of 6.005 Elements of Software Construction. The official MIT Subject listing describes 6.005 as an introductory class to software development, with an emphasis on learning how to design and write good software. The official course description [18] is as follows:

> "Introduces fundamental principles and techniques of software development, i.e., how to write software that is safe from bugs, easy to understand, and ready for change. Topics include specifications and invariants; testing, test-case generation, and coverage; state machines; abstract data types and

> representation independence; design patterns for object-oriented programming; concurrent programming, including message passing and shared concurrency, and defending against races and deadlock; and functional programming with immutable data and higher-order functions."

In the Fall 2011 semester of 6.005, Collabode was used in 13 recitations ranging from 10-30 students per session. One instructor was in charge of each recitation, with six instructors in total. Teaching assistants who served as recitation instructors from that semester were asked to user test CollabodeTA.

Although the CollabodeTA mobile instructor interface consists of a number of separate components, evaluation was done on the system as a whole. Three users who were TAs/recitation instructors from the semester of 6.005 from which Collabode was used were chosen to do the user study. Users were given a statement of the goal of the mobile instructor interface and a list of tasks and questions to consider while using the system before being presented with the interface. These questions were chosen to gauge whether CollabodeTA indeed achieve its goals of making it easier for instructors to gauge individual and overall student progress and performance earlier and more easily than it was possible to do before, without intruding on the instructor's natural teaching style. The questions were as follows:

Goals

> Gain insight on what goes on during a software lab session that you couldn't achieve without the perspective of the mobile lab instructor interface.

Tasks/Things to think about

Where is the majority of the class in the assignment? Estimate how much of the assignment(s) the class is on track to finishing.

1. What key points in the recitation material are getting across and which are not being understood?
2. Which, if any, students are significantly ahead or behind, and why? Is anyone done?
3. If no one were on the help queue, which students would you help right now? Do the students on the help queue correspond with those that appear to need the most help?
4. What, if any, common problems does it look like people are running into? Does the interface help make this clear? Does it coincide with what you remember about the recitation when you taught it?

Reflection

1. How easy was it to achieve each of the tasks?
2. Was the overall goal achieved?
3. Do you have any other suggestions or thoughts?

The pre-recorded recitations chosen for user evaluation were chosen based on expected console output format, diversity of assignments, and when in the semester the recitation occurred. We chose assignments whose expected primary console output was test-driven, so that the console output view would be the most meaningful and useful to instructors. This view was chosen because it was the richest view of student performance implemented in CollabodeTA and the one that we wished to get the most feedback on. Within the constraints of the console output format, we still chose a diversity of assignments, many of which resulted in output that was still test driven but either partially nondeterministic or had varying amounts and types of student errors and arbitrary or debug output. Finally, we chose assignments from three different times during the semester. The earliest assignments shown in user tests were from the first recitation, where the majority of students were just getting familiar with the Java programming language and therefore were completing very structured, discrete tasks, and the latest was from the last recitation, where students were assumed to be proficient in Java and moved on to tackling more conceptually difficult tasks.

Students achieved correctness for each of these assignments by printing the correct output to their consoles, but were free to print intermediary debug statements as well. In the end, the following recitation and assignments were chosen:

Recitation 1

Recitation 1 consisted of a series of short assignments to help students become familiar with simple Java syntax, operations, and data structures. Our users, who were familiar with the assignments though not necessarily of the specific students in the recitations replayed for them, were shown Recitation 1 data at four evenly-spaced points in time during the time span of the recorded recitation session, and were asked to comment on anything they noticed as the recitation session progressed and answer the questions given to them as best as they could.

Financial Calculations

Students were asked to make a series of financial calculations such as calculating the expected balance after a certain amount of time given a starting balance and annual interest rate. Correctness was verified by values printed to standard output.

Equals Checking

Students were asked to check whether the contents of two arrays were equal. Correctness was verified by four tests, each of which printed CORRECT or INCORRECT to standard output along with a description of the test.

Prime Checking

Students were asked to implement three increasingly efficient versions of a prime number checker. Correctness was verified by the result of the prime checker as well as the runtime of each prime number checker, both of which were printed to standard output.

Palindrome

Students were asked to write a program that verified whether given strings were palindromes. Correctness was verified by printing the correct value to standard output.

Drawing Cards

Students were asked to simulate the process of drawing playing cards from a deck with and without replacement between each draw, where cards were represented as objects in a Java array representing a deck of cards. Correctness was verified by printing the contents of the correct final arrays to standard output.

Recitation 10

IntensiveSumRC.java

Students were asked to write a multithreaded program to sum the elements of an array. Correctness was achieved by printing the correct sum to standard output.

Recitation 13

Threading

Students were asked to find and eliminate a race condition in a small multithreaded program that caused occasional inconsistencies in text printed to standard output. Correct output was assumed when each of 10 threads consistently printed a complete, identical statement to standard output without races.

Results

Response to CollabodeTA was positive. All of our user study participants felt that the console output view would be useful, and liked the idea of viewing class performance as an aggregate within the class layout view. They felt that the tablet form factor was non-intrusive in their workflow, and were able to answer the questions given to them at the start of the user study, which they all agreed would have been difficult or impossible to do without CollabodeTA.

All three of the user study participants suggested features that would make the interface more useful for them, most of which we had preconceived and were simply not included in the first iteration of CollabodeTA. Most of these related to being able to drill down into student code more easily. The main confusion points with the interface were with specific design choices in how cluster diffs are displayed in the console output view, and in which aspects of the interface linked back to other parts of Collabode that they were familiar with. These suggestions have been incorporated into the section on future work (Section 5.1).

4.2 Recitation Studies Using CollabodeTA

In developing CollabodeTA, where our focus was initially to aid instructors helping students in real time, we realized that the data we have collected is useful for analyzing recitations or labs after they finish. Specifically, the cluster data we have for each student and assignment over time within each recitation session, in addition to other metrics that could be easily added to the interface, is valuable information for determining factors such as how difficult the assignments were, which tasks were most difficult for students and how long each of them took to complete, which concepts were most easily grasped and which caused the most confusion, and so on.

Simply by analyzing cluster data on the first recorded recitation session, we found that students ran their code relatively infrequently compared to how much time they spent writing it, and often did not spend much time iterating on each task. This, of course, applies to the recitation-scale exercises we had access to, and by analyzing other metrics we collect through the interface, CollabodeTA can begin to give us a much richer view of how to better design future recitations and lab sessions.

Chapter 5

Conclusion

We found that the process of writing code in class in the context of software education is more opaque to instructors than it could be in terms of understanding the pace and amount of student progress and understanding on in-class assignments. It is also difficult to accurately gauge student performance on lab exercises for the purposes of identifying those that need help, or common mistakes that should be addressed for the entire class.

CollabodeTA was built on top of the Collabode real-time collaborative web IDE to address this problem by taking advantage of the the real-time, keystroke-by-keystroke data provided by Collabode to act as a tool to aid software lab instructors during lab sessions. Through it, we provide a set of views for lab instructors to address and locate students queued for help, and gain insight on how students are performing on tasks while in the context of a lab session. User testers who were previously TAs of the MIT 6.005 Elements of Software Construction class were able to use CollabodeTA effectively to identify students who were on task as well as outlier problems that they would have addressed had they known they were an issue at the time. CollabodeTA is a web application that can be accessed from any Javascript-enabled browser connected to a Collabode server, and is optimized for tablet devices for portability within the classroom.

CollabodeTA illustrates a new use case for the Collabode as a mobile instructor interface, and shows promise both as a new model of collaboration that supports software development and programming education in a lab setting and as an evaluation tool for current coursework that can contribute to improving the quality of software and programming classes in the future.

5.1 Future Work

In this section, we suggest ways in which CollabodeTA can be improved and extended to become an even more useful tool for software lab instructors.

5.1.1 New Metrics for Student Activity and Progress

Analyzing console output gives us a rich yet limited look on in-class student performance. In order to provide as comprehensive of a view of the classroom as possible for software lab instructors, CollabodeTA should be extended to measure student progress, understanding, and performance by more of the metrics presented in Section 3.3, notably:

- Integration with a unit test framework such as JUnit [25], especially if applicable to course content, which would provide similar functionality to our test function output analysis without the noise of spurious print statements
- Compiler metrics: What kinds of problems did students encounter and for how long before they even ran their code?
- Additional run-time metrics beyond identifying exceptions and clustering on console outputs. In particular, identifying non-fatal or non-terminating errors is a challenging but interesting task that could be attempted.
- Perhaps the biggest contribution Collabode makes is giving us insight into what and where students are editing at any given moment. There are a wealth of student-produced metrics that could be collected in order to give instructors a more

complete idea of how any given student or the class as a whole is doing on given assignments.

- Similarly, Collabode itself performs operations on top of student keystrokes and actions in the Collabode interface that can be collected. The suggestion in Section 3.3 was to collect integration metrics, as a measure of the quality of code each student is writing, not with respect to logic, but rather with respect to how correct it is in terms of syntax and organization.

While measuring some of these metrics may take some creativity and more effort than others to collect efficiently, all of these can be collected from Collabode. The framework we have established in our implementation of CollabodeTA is also flexible and easy to extend with new data while being minimally intrusive on the main Collabode system as it is running.

5.1.2 Student Cards

Student cards are meant to be decorated with more at-a-glance indicators, especially as other metrics are added. Examples of these are:

- Additional **badges** and **labels** indicating metrics such as the amount of code written, where in the code the student is currently working, which files have been touched, and so on. Even the current help queue badge may be modified to reflect either a student's queued/helped status or their position on the queue.
- **Progress bars or grids** corresponding to tests passed in a unit test suite (if applicable), or by any new measure of progress

- **Color overlays or subtle animations** such as glowing or gentle pulsing to bring the instructor's attention to students that the system identifies as being in need of it.

According to our original design motivations and user studies, we believe that the class layout view, which displays all student cards at once, will benefit immensely from additional student card indicators if they are added tactfully.

5.1.3 Class Layout View

The primary barrier to use in the class layout view is a way to create spatial arrangements of student cards with minimal human effort. We are certain that this burden should not be laid on the lab instructor, especially as class sizes may be upwards of 20 students. Because students at MIT classes are rarely (if ever) asked to work on designated machines in designated locations, we cannot assume class layout to remain consistent between software lab or recitation sessions. However, the classroom in which the lab or recitation takes place is generally consistent throughout the academic term. The 6.005 Karga queue asks students to provide a short description of where they are sitting, which we could also implement in our system, perhaps by having students identify their location on a map of the classroom at the beginning of the classroom, which would provide a reasonably accurate indicator of their location.

A second interesting area for future work within the class layout view is the problem of optimizing student card sizes for arbitrarily large-sized classes. Our interface is currently optimized for class sizes of 30-50 students, after which student cards would begin to obscure each other, especially when arranged in a spatial layout. The dimensions of student cards could in hypothetically be determined formulaically, to minimize the amount of overlap between cards in large classes while preserving legibility.

5.1.4 Console Output View

The console output view is an exciting start to collecting student performance metrics. The following suggestions for future work arise from our user study feedback and own personal experience as instructors in software engineering and programming classes.

Improved cluster normalization

An optimization for future consideration is to implement fuzzy matching during the normalization process such that inconsequential differences are ignored, such as differences in line number for the same code across different implementations. A more sophisticated optimization is to automatically detect non-deterministic output and also remove these differences when generating the canonical form of an output.

Improvements to cluster diffs

We have implemented small optimizations to the rendering of cluster diffs, but we believe that this can be improved further, make sure the points that differ between each cluster are immediately apparent to users.

Smoother integration with Collabode and CollabodeTA

Various aspects of the console output view can and should be linked back to relevant pieces of Collabode and the CollabodeTA mobile instructor interface. This includes linking from student names back to their student panels, linking from exceptions or console output to relevant pieces of code, and the ability to view previous outputs the student has encountered during the same session, to compare with their most recent output.

In addition to linking, there is no reason why student card thumbnails shown the console output view should not also show a limited set of indicators as the rest of CollabodeTA does on student cards. An example of this is highlighting students on the help queue from within the console output view. This would make questions such as "does this student who has been getting a particular exception for several minutes want help? Are they on the queue?" much easier to answer.

Scrolling through time

We have shown that we can replay recitations and stop at arbitrary time points in recorded recitation data. From our user studies, we also found that instructors would find useful the ability to scroll through time *during* the course of the recitation, in order to view the class or a student's progress history on the spot. The primary difficulty in this task is finding a way to implement it that is performant enough to do in real time. The cluster precomputation we did when running prerecorded data is an example of an optimization that would most likely be necessary to implement real-time scrolling.

Displaying long output

We have mentioned that we truncate code containers for clusters when the cluster text becomes unreasonably long to display in its entirety. Our solution for this was to implement scrolling in the inner code containers; however, a better or more intuitive solution may be found, perhaps including simply providing an affordance for expanding a container holding truncated output. More likely, in cases where long output is not necessarily due to error, it would make sense to scroll all code containers in lockstep, so that an instructor can view the same line of output for every cluster variation at once without having to scroll each code container individually to the same spot.

Overlay view

The overlay view is a reimagined concept for test-driven output in the console output view that instead of displaying diffs between clusters and a prototype cluster, stacks all outputs on top of each other *intelligently*, and allows instructors to see using an overlay paradigm where student outputs differ. We make the distinction that overlays must be done “intelligently” because our prototype implementation of this showed that the variations in console output are too many for naive overlays, and at the very least, algorithms for entering whitespace and line breaks within clusters in order to maximally align them must be developed before this view is feasible. However, if implemented correctly, this has potential to be an interesting way for instructors to select pieces of erroneous cluster text and drill into what the cause might have been.

5.1.5 Student Panels

Similarly to student cards, more information should be added to student panels as other metrics are added. In particular, our user studies indicate that instructors would find it helpful if details collected from CollabodeTA metrics linked to the relevant files, code snippets, or consoles in Collabode.

5.1.6 Help Queue

While the CollabodeTA help queue implementation is designed to be simple, small improvements can be made to its interface and functionality to increase the transparency of its state for all users, both instructors and students. In particular, students should be able to see their position on the queue, and have the flexibility to remove themselves from it if they wish. In the case of multiple instructors, they should also be able to see which instructor is helping which student on the queue at any given time. Because student info panels and other views can be opened while the help queue is shown, we also envision the queue to be a way of partitioning and assigning students between multiple instructors, to streamline the answering process by grouping together students with similar questions.

References

[1] *Collabode*. [Online]. Available: `http://groups.csail.mit.edu/uid/collabode/`

[2] M. Goldman, G. Little, and R. C. Miller, "Real-Time Collaborative Coding in a Web IDE". In *UIST*, 2011.

[3] M. Goldman, G. Little, and R. C. Miller. "Collabode: Collaborative Coding in the Browser". In *CHASE*, 2011.

[4] M. Goldman. "Thesis Proposal: All the program's a stage, and all the programmers merely players". [Online]. Available: `http://people.csail.mit.edu/maxg/proposal/maxg-proposal.pdf`

[5] *iTalc - Intelligent Teaching and Learning with Computers*. [Online]. Available: `http://italc.sourceforge.net`

[6] *TEAL - Technology-Enabled Active Learning*. [Online]. Available: `http://icampus.mit.edu/teal/`

[7] K. Koile, and D. Singer, "Development of a Tablet-PC-based System to Increase Instructor-Student Classroom Interactions and Student Learning". In *The Impact of Pen-based Technology on Education; Vignettes, Evaluations, and Future Directions*. Berque, D., Gray, J., and Reed, R. (editors). Purdue University Press, 2006.

[8] *Cloud9 IDE*. [Online]. Available: `http://cloud9ide.com/`

[9] *Ajax.org Cloud9 Editor (Ace).* [Online]. Available: `http://ace.ajax.org/`

[10] *eXo Cloud IDE*. [Online]. Available: `http://cloud-ide.com/`

[11] *ShiftEdit*. [Online]. Available: `http://shiftedit.net/`

[12] *WaveMaker*. [Online]. Available: `http://www.wavemaker.com/`

[13] *WWWorkspace*. [Online]. Available: `http://www.willryan.co.uk/WWWorkspace/`

[14] *JQuery Mobile*. [Online]. Available: `http://jquerymobile.com/`

[15] *JQuery UI*. [Online]. Available: `http://jqueryui.com/`

[16] MIT, *6.004 help queues*. [Online]. Available: `https://courses.csail.mit.edu/6.004/queues/`

[17] MIT, *6.005 Karga queue.* [Online]. Available: `https://karga.csail.mit.edu/karga/`

[18] MIT, *Subject Listing and Course Catalog*. [Online]. Available: http://student.mit.edu/catalog/search.cgi?search=6.005&style=verbatim

[19] *Less.js*. [Online]. Available: http://lesscss.org

[20] *JQuery UI for iPad and iPhone*. [Online]. Available: http://code.google.com/p/jquery-ui-for-ipad-and-iphone/

[21] Google, *diff-match-patch*. [Online]. Available: http://code.google.com/p/google-diff-match-patch/

[22] *iScroll*. [Online]. Available: http://cubiq.org/iscroll-4

[23] *Flot*. [Online]. Available: http://code.google.com/p/flot/

[24] S. Levithan, *datetime format*. [Online]. Available: http://blog.stevenlevithan.com/archives/date-time-format

[25] JUnit. [Online]. Available: http://junit.org/