San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Fall 12-15-2016

Web-based Integrated Development Environment

Hien T. Vu
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Databases and Information Systems Commons, and the Graphics and Human Computer Interfaces Commons

Recommended Citation
Vu, Hien T., "Web-based Integrated Development Environment" (2016). *Master's Projects*. 499.
DOI: https://doi.org/10.31979/etd.f6v3-wqu3
https://scholarworks.sjsu.edu/etd_projects/499

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Web-based Integrated Development Environment

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Hien T. Vu

December 2016

© 2016
Hien T. Vu
ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Web-based Integrated Development Environment

by

Hien T. Vu

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2016

Cay Horstmann	Department of Computer Science
Ronald Mak	Department of Computer Engineering
Thomas Austin	Department of Computer Science

ABSTRACT

Web-based Integrated Development Environment

by Hien T. Vu

As tablets become more powerful and more economical, students are attracted to them and are moving away from desktops and laptops. Their compact size and easy to use Graphical User Interface (GUI) reduce the learning and adoption barriers for new users. This also changes the environment in which undergraduate Computer Science students learn how to program. Popular Integrated Development Environments (IDE) such as Eclipse and NetBeans require disk space for local installations as well as an external compiler. These requirements cannot be met by current tablets and thus drive the need for a web-based IDE. There are also many other challenges of moving a desktop-based IDE to a web-based one.

There are many web-based IDEs currently in development. However, this project focuses on four particular open-sourced web-based IDEs: Ace, CodeMirror, ICEcoder and CloudCoder. Ace, CodeMirror and ICEcoder are web-based code editors and CloudCoder is a complete web-based exercise system. All of them were found to be integrable with CodeCheck except for ICEcoder.

The CloudCoder integrated Codecheck was deployed for three classes during the Fall 2016 term at San Jose State University. Empirical data showed that students had a better grasp of the subject matter when exposed to exercises hosted by the enhanced CodeCheck. This was measured indirectly via the scores from projects, quizzes and exams.

ACKNOWLEDGMENTS

I would like to thank Professor Cay Horstmann for giving me an opportunity to work on this project. I would also like to thank Professors Ronald Mak, Thomas Austin and Fabio Di Troia for allowing their students to participate in testing the enhanced CodeCheck and for allowing me to collect data to measure the effectiveness of this system in improving the students' performance. Last but not least, I would like to thank Professor Sami Khuri and the Department of Computer Science for the financial support of the virtual machines.

TABLE OF CONTENTS

CHAPTER

1 **Introduction** 1

2 **Prior work** 3

3 **Technologies Used in the Project** 6

3.1 CodeCheck 6

3.2 Web-based code editors 15

3.2.1 Ace 15

3.2.2 CodeMirror 16

3.3 CloudCoder 18

4 **Integration of Ace and CodeMirror with CodeCheck** 27

5 **Integration of CloudCoder with CodeCheck** 32

5.1 Data model 33

5.2 Views 34

5.2.1 Edit problem page 34

5.2.2 Development page 35

5.3 Webapp and CodeCheck communication 39

6 **Results** 43

6.1 Enhanced CodeCheck Deployment 43

6.1.1 CS49J - Programming in Java 43

6.1.2 CS152 - Programming Paradigms 45

6.1.3 CMPE180 - Data Structures and Algorithms in C++ 48

6.2 Enhanced CodeCheck Evaluation 50

7 Deployment Challenges 56

8 Conclusion 59

APPENDIX

A CloudCoder Compilation and Installation 63

A.1 Prerequisites 63

A.2 Compiling CloudCoder 63

A.3 Deploying CloudCoder 64

B Source Code 68

LIST OF FIGURES

1	CodeCheck - Problem layout	6
2	CodeCheck - Upload problem interface	7
3	CodeCheck - Upload problem selected	7
4	CodeCheck - Uploaded problem submitted	8
5	CodeCheck - Student interface for a single-file problem	9
6	CodeCheck - Student interface for a multi-file problem	10
7	CodeCheck - Report with compilation errors of a single-file problem	11
8	CodeCheck - Report with compilation errors of a multi-file problem	12
9	CodeCheck - Report with test failing for the simple problem	13
10	CodeCheck - Report with test passing for the multi-file problem	14
11	Ace example	15
12	CodeMirror example	17
13	CloudCoder - System	18
14	CloudCoder - Login page	19
15	CloudCoder - Student main page	20
16	CloudCoder - Student problem page	20
17	CloudCoder - Instructor main page	21
18	CloudCoder - User management page	22
19	CloudCoder - Exercises management page	22
20	CloudCoder - Exercise composer (1 of 6)	23
21	CloudCoder - Exercise composer (2 of 6)	24

22 CloudCoder - Exercise composer (3 of 6) 24

23 CloudCoder - Exercise composer (4 of 6) 25

24 CloudCoder - Exercise composer (5 of 6) 25

25 CloudCoder - Exercise composer (6 of 6) 26

26 Codecheck - Enhanced uploaded problem page 27

27 Codecheck - Instructor interface for a simple problem 28

28 Codecheck - Instructor interface for a simple problem 29

29 Codecheck - Enhanced student interface for a simple problem . . 30

30 CodeCheck - Enhanced system 32

31 CodeCheck - Enhanced exercise composer (1 of 6) 36

32 CodeCheck - Enhanced exercise composer (2 of 6) 36

33 CodeCheck - Enhanced exercise composer (3 of 6) 37

34 CodeCheck - Enhanced exercise composer (4 of 6) 37

35 CodeCheck - Enhanced exercise composer (5 of 6) 38

36 CodeCheck - Enhanced exercise composer (6 of 6) 38

37 CS49J - Average number of attempts 44

38 CS49J - Average number of minutes per attempt 44

39 CS49J - Average project scores 45

40 CS152 - Average number of attempts 46

41 CS152 - Average number of minutes per attempt 46

42 CS152 - Exam scores 47

43 CS152 - Quiz scores 48

44 CS152 - Quiz scores filtered 49

45 CMPE180 - Average number of attempts 49

46 CMPE180 - Average number of minutes per attempt 50

47 CMPE180 - Exam score 51

48 CMPE180 - Quiz scores distribution 52

49 Questions on the effectiveness of the enhanced CodeCheck 53

50 Questions on the usability of the enhanced CodeCheck 54

51 Questions on the applicability of the enhanced CodeCheck 55

CHAPTER 1

Introduction

In an introductory or an intermediate programming course, students are encouraged to practice writing programs in order to master the language and its syntax. For this, the common recommendation to students is to install an Integrated Development Environment (IDE). With some IDEs, there is an additional requirement to install a compiler and linker. There are many IDEs available [19, 20] and the most widely known open-source ones are NetBeans and Eclipse. The majority of these IDEs are intended to be used on desktops or laptops and certainly not intended for tablets and other small portable devices. This is because local storage is limited and compilers are not readily available for their architectures.

In this project, four open-sourced web-based IDEs (Ace, CodeMirror, ICEcoder and CloudCoder) were evaluated for their capabilities. Ace, CodeMirror and ICEcoder are web-based code editors. These can serve as the front-end of an IDE. To complete an IDE, they need to be integrated with CodeCheck. ICEcoder is a source file management and code editing tool [13] and uses CodeMirror for its editors [14]. ICEcoder has only a 14-day trial period so it was not evaluated further in this project. CloudCoder is a complete IDE in that it has a code editor and a builder. It is also an exercise system like CodeCheck where submissions are automatically graded. Unlike CodeCheck, CloudCoder supports individual student workspace and provides the capability to group exercises and students by classes.

This project was split into two phases: 1) evaluate Ace and CodeMirror and 2) evaluate CloudCoder. Both phases also include the integration of the software

with CodeCheck. Chapter 3 gives an introduction to CodeCheck, Ace, CodeMirror and CloudCoder. The integration of the web-based code editors and CloudCoder are discussed in Chapters 4 and 5 respectively. Chapter 6 discusses in detail the deployment of the enhanced CodeCheck and the results obtained from the three classes that participated in its evaluation.

CHAPTER 2

Prior work

An automated web-based grading system called Infandango was developed by Hull, Powell and Klein [12]. The system allows students to submit Java source files and a backend JUnit test engine compiles and executes a set of predefined tests. The outcome is then stored in a database and communicated back to the students. The Infandango system is composed of four components: a web front-end, the CoSign authentication module, the Jester JUnit tester and the PostgreSQL database. Based on the authors' conclusion, the components are loosely coupled and they can be swapped out with other appropriate components.

Web-CAT, developed by Edwards and Perez-Quinones [5], also provides an automatic web-based grading system. It supports mainly Java and C++ exercises. Web-CAT exercises can be configured to require the test cases together with the source files. This is one of the well-known features of Web-CAT. Web-CAT is extensible with its plug-ins [4]. Plug-ins can be developed to support other programming languages and they can be configured to collect more statistical data from the students' submissions.

Pritchard's approach with Websheet is for an instructor to setup a solution and provide the locations of the "fill-in-the-blank" areas [18]. Through a browser, students provide the fill-in and submit for evaluation. Websheet supports both Java and C++ and uses CodeMirror for its text editor. It is designed for in-class exercises or practice homework problems. After three failed attempts, the solution is made available to the user.

Deeb and Hickey [3] developed Spinoza, which is also a web-based IDE with an automatic grading engine. With SpinozaExercises, instructors can quickly create a problem by providing a description and optional skeleton codes. SpinozaHomeworks are more comprehensive in that solution files and test cases are required so they can be used to evaluate submitted solutions. A unique feature of Spinoza is the share mode where instructors can display submitted code to the entire class. The instructor view offers instructors the current progress of the class. In real time, the instructors can see the students that have completed the assignment, those that still have syntax errors and those that have the same equivalent class file. A survey was done by the authors with 238 students and found that 36% preferred this system over lectures with Powerpoint. Also, 28% of the students thought Spinoza is as good as a learning method as Powerpoint lectures.

Collabode from Goldman, Little and Miller [7] enables programmers to synchronously collaborate and immediately share the changes with one another. The web-based development environment is powered by Eclipse on the server side. Each user accesses the files via their respective browsers. The multiple editors are supported by EtherPad, an open-source collaborative online editor [6], which shares the changes in near real-time. The system supports the Python and Java programming languages. The authors describe three collaborative scenarios: micro-outsourcing, test-driven paired programming and mobile instructors. With micro-outsourcing, many programmers can make small contributions to a developer. And in the last scenario, instructors can help the students by connecting to their IDEs.

Kurtz, Fenwick, Tashakkori, Esmaili and Tate [16] introduced the Code Magnet microlabs into their lecture. Students use tablets during lectures to graphically compose solutions to in-class conceptual problems. The submitted solutions are eval-

uated by Code Magnet and feedback is provided. A control study with two different groups of students was done at SUNY Stony Brook where three identical questions were asked on their tests. The questions were taken from lectures on the binary tree traversals, the building of binary search tree (BST) from postorder traversal and the building of general binary tree from preorder and inorder traversals. The authors found that the group that had been exposed to microlabs performed, on average, better than those that had not been by 8 to 10% with a statistical significance of $p = 0.038$.

CHAPTER 3

Technologies Used in the Project

3.1 CodeCheck

As described by the developer, CodeCheck is a "convention over configuration" service for grading of programming assignments [9]. It is a web-based system where programming problems can be assigned and users' solutions will be validated automatically. This is intended to be used in an introductory programming class where students can learn by trial-and-error. The system does not track users' progress nor does it have any identification information about the users. It does provide a signed zip file containing the solution files and its validation report. As an assignment, students can turn in this zip file to the instructor.

A CodeCheck problem consists of a description file, source files to be completed, source files for helper or tester classes, input files and solution source files. The files are laid out according to Figure 1 [9].

```
problem.html (optional)
student/
    Source file(s) to be completed (if needed)
    Source files for helper or tester classes (if needed)
    Input files (if needed)
solution/
    Source file(s)
```

Figure 1: CodeCheck - Problem layout

The *student* and *solution* are fixed directory names. The description of the problem is optional and is expected to be in HTML format. CodeCheck recognizes the description file if its name is *problem.html*. The *student* directory contains the files with partial code that the students will start with. All of the files in this directory

will be displayed by CodeCheck except for the input file. The corresponding solution files for the problem are stored inside the *solution* directory. Only files that are in both directories will be made editable by CodeCheck for the students. All of the files and directories need to be zipped before they can be uploaded onto CodeCheck.

As shown in Figure 2, the instructor uses CodeCheck's web page interface to upload a problem zip file.

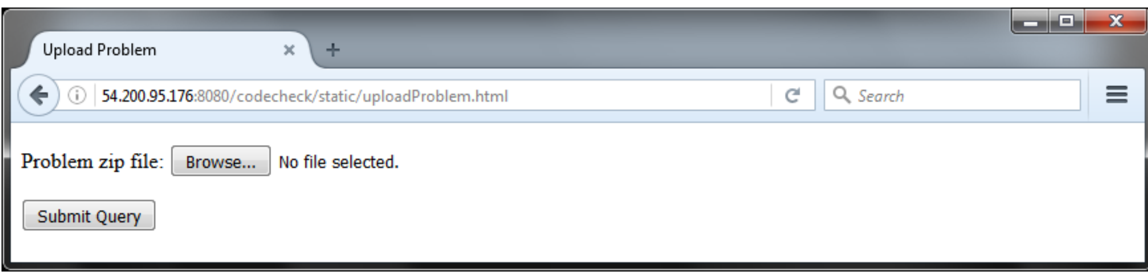


Figure 2: CodeCheck - Upload problem interface

By clicking on the 'Browse' button, a file browser dialog box will appear. The instructor can then navigate the file system to find the problem zip file. Figure 3 shows an example simple.zip file ready to be uploaded.

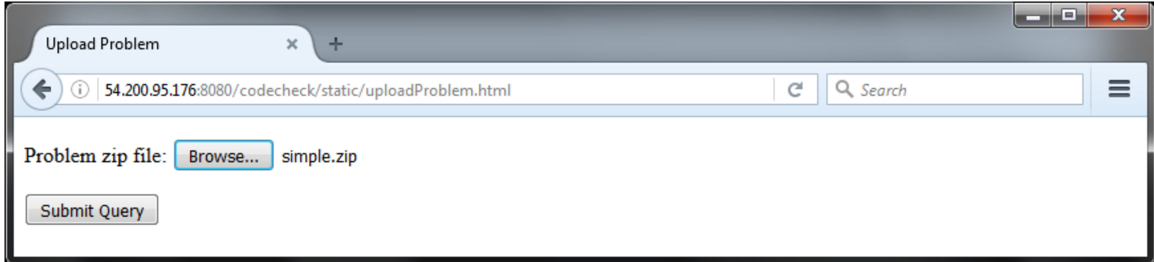


Figure 3: CodeCheck - Upload problem selected

Upon clicking on the 'Submit Query' button, a new page containing a randomly generated uniform resource locator (URL) is displayed as shown in Figure 4.

This URL can be given to students to access their programming assignment. The 'Preview' link takes the instructor to the generated URL. It can be used to test out

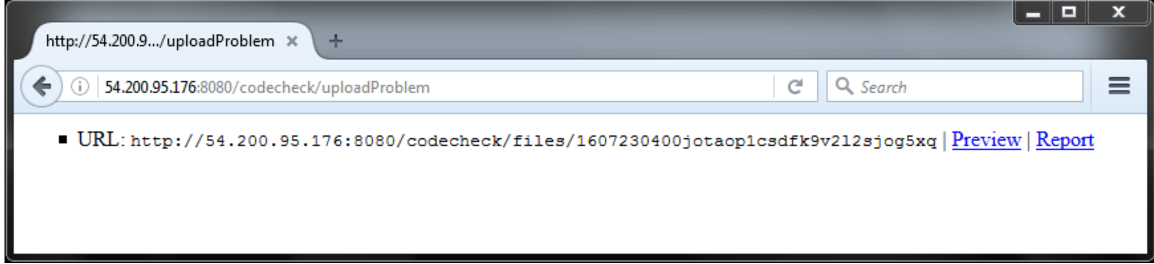


Figure 4: CodeCheck - Uploaded problem submitted

the student's interface. The 'Report' link takes the instructor to a validation report. This report is generated using the solution source files. The instructor can use this report to make sure that the solution files are correct. If there are issues that need correction, a new zip file can be uploaded to CodeCheck.

Figures 5 and 6 show the starting web page of problems with single and multiple files, respectively. Each source file in the *student* directory is displayed in an individual HTML `textarea`. In these two examples, the *problem.html* file contains the text in between and including the horizontal bars above the "Complete the following file" line. Students can directly make changes to the code in the text-area and click on the 'Submit Query' button to submit their solution.

Upon submission, CodeCheck compiles the submitted files. If there are errors, the report displays the compiler's error messages. Examples of the compilation errors are shown in Figure 7 and 8. These errors are taken directly from the compiler. The error in Figure 7 is a missing open curly bracket on line 5. There are two missing semicolons in Figure 8 but only one is caught. This is because the compiler stops upon detecting the missing semicolon on line 16 of the second source file. Once that is fixed, the second missing semicolon on line 4 of the first file will be detected. To fix these errors, the student can use the 'Go back' button on the browser to make correction and then submit again. These steps can be repeated until a successful

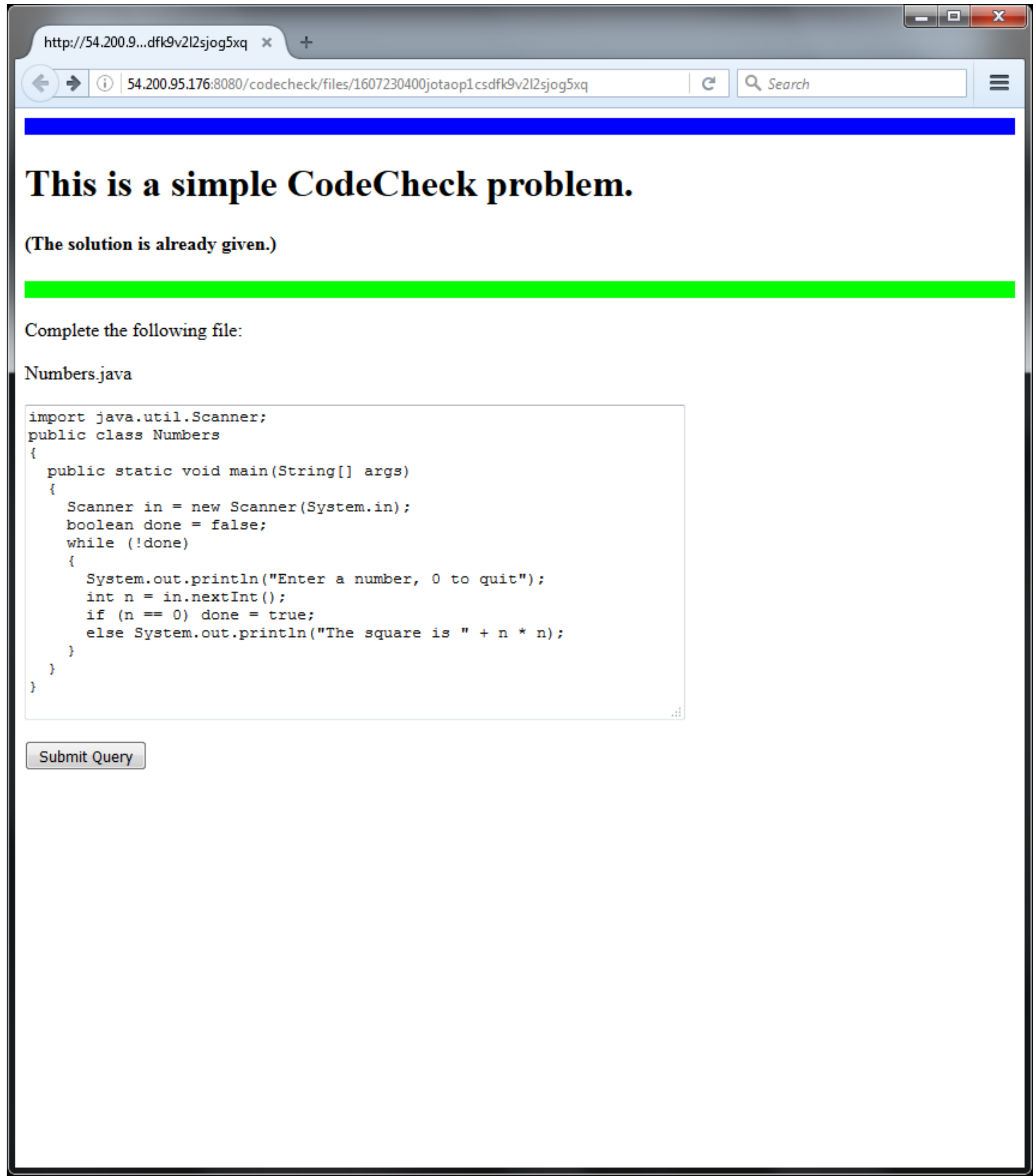


Figure 5: CodeCheck - Student interface for a single-file problem

compilation.

The compiled codes are tested with the test input from the input files. If the test fails, then a report is given as shown in Figure 9. Using the solution files, CodeCheck

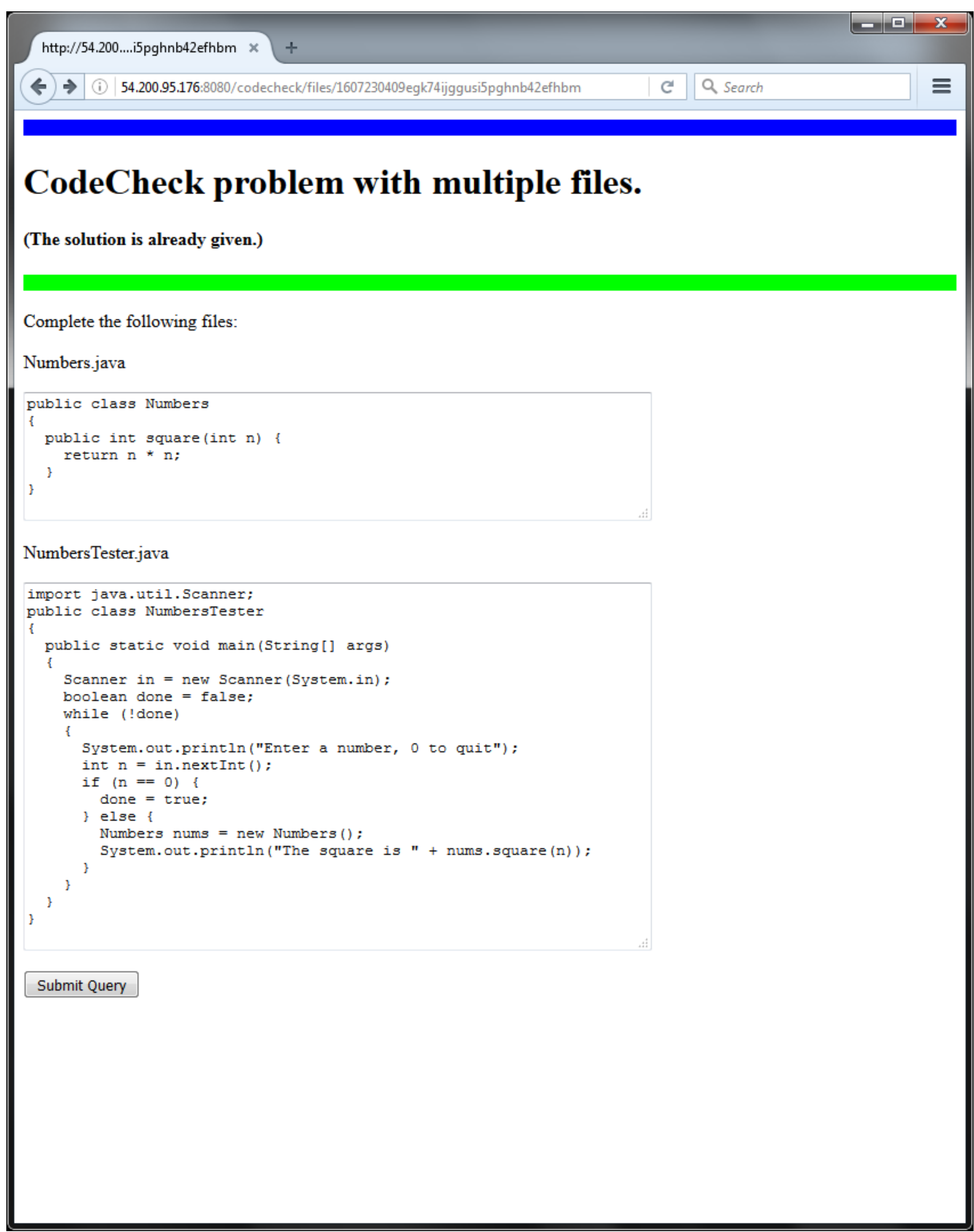


Figure 6: CodeCheck - Student interface for a multi-file problem

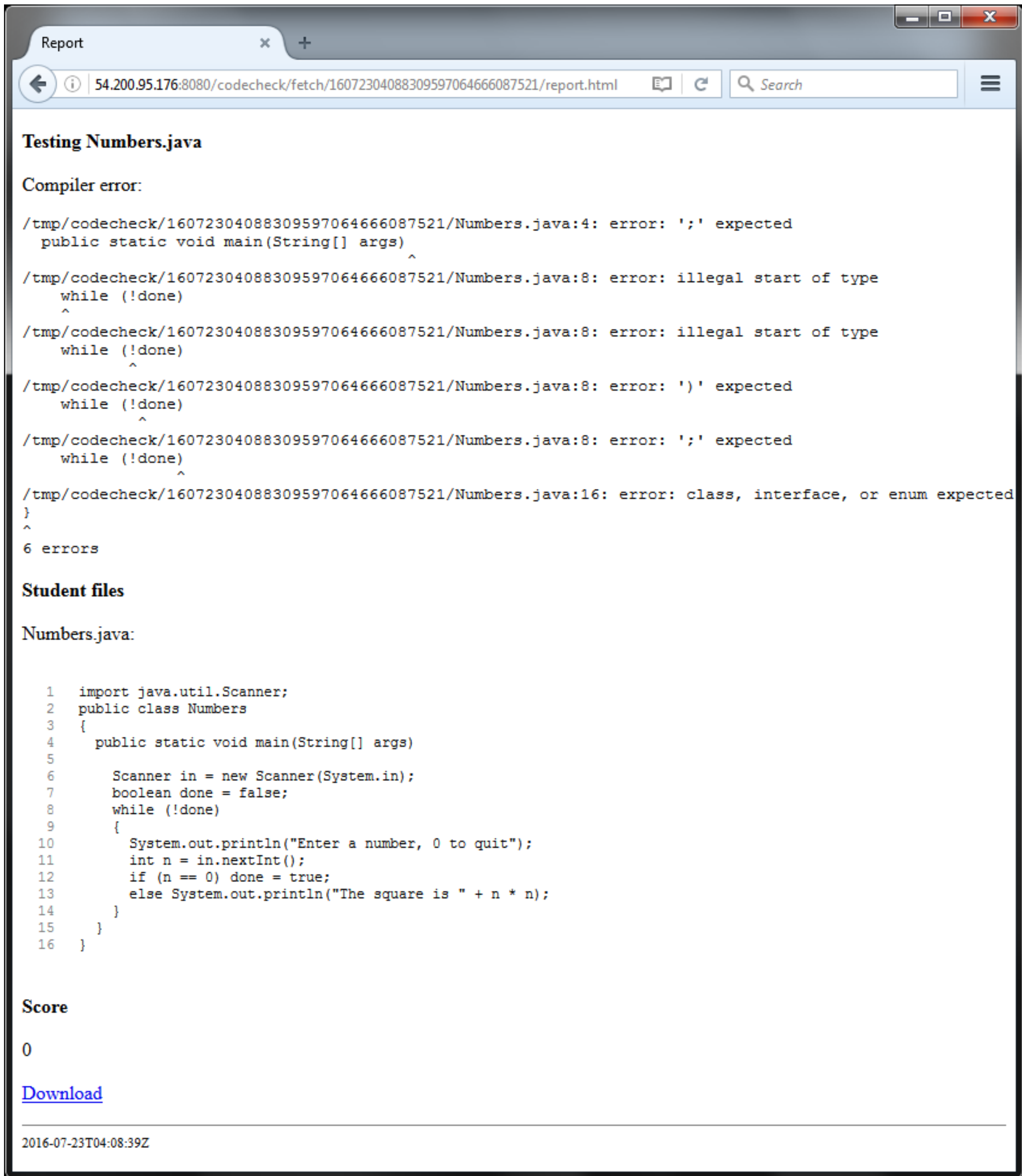


Figure 7: CodeCheck - Report with compilation errors of a single-file problem

also provides the expected output in the report to the student. CodeCheck compares the output from the student's and the solution's source files so an exact match of the format of the output is expected. The student can again go back one page on the

Report

54.200.95.176:8080/codecheck/fetch/16072304164883230728572355701/report.html

Search

Testing NumbersTester.java

Compiler error:

```
/tmp/codecheck/16072304164883230728572355701/NumbersTester.java:16: error: ';' expected
        System.out.println("The square is " + nums.square(n))
                                                             ^
1 error
```

Student files

Numbers.java:

```
1   public class Numbers
2   {
3     public int square(int n) {
4       return n * n
5     }
6   }
```

NumbersTester.java:

```
1   import java.util.Scanner;
2   public class NumbersTester
3   {
4     public static void main(String[] args)
5     {
6       Scanner in = new Scanner(System.in);
7       boolean done = false;
8       while (!done)
9       {
10        System.out.println("Enter a number, 0 to quit");
11        int n = in.nextInt();
12        if (n == 0) {
13          done = true;
14        } else {
15          Numbers nums = new Numbers();
16          System.out.println("The square is " + nums.square(n))
17        }
18      }
19    }
20  }
```

Score

0

Download

2016-07-23T04:16:10Z

Figure 8: CodeCheck - Report with compilation errors of a multi-file problem

browser to update the source file and re-submit.

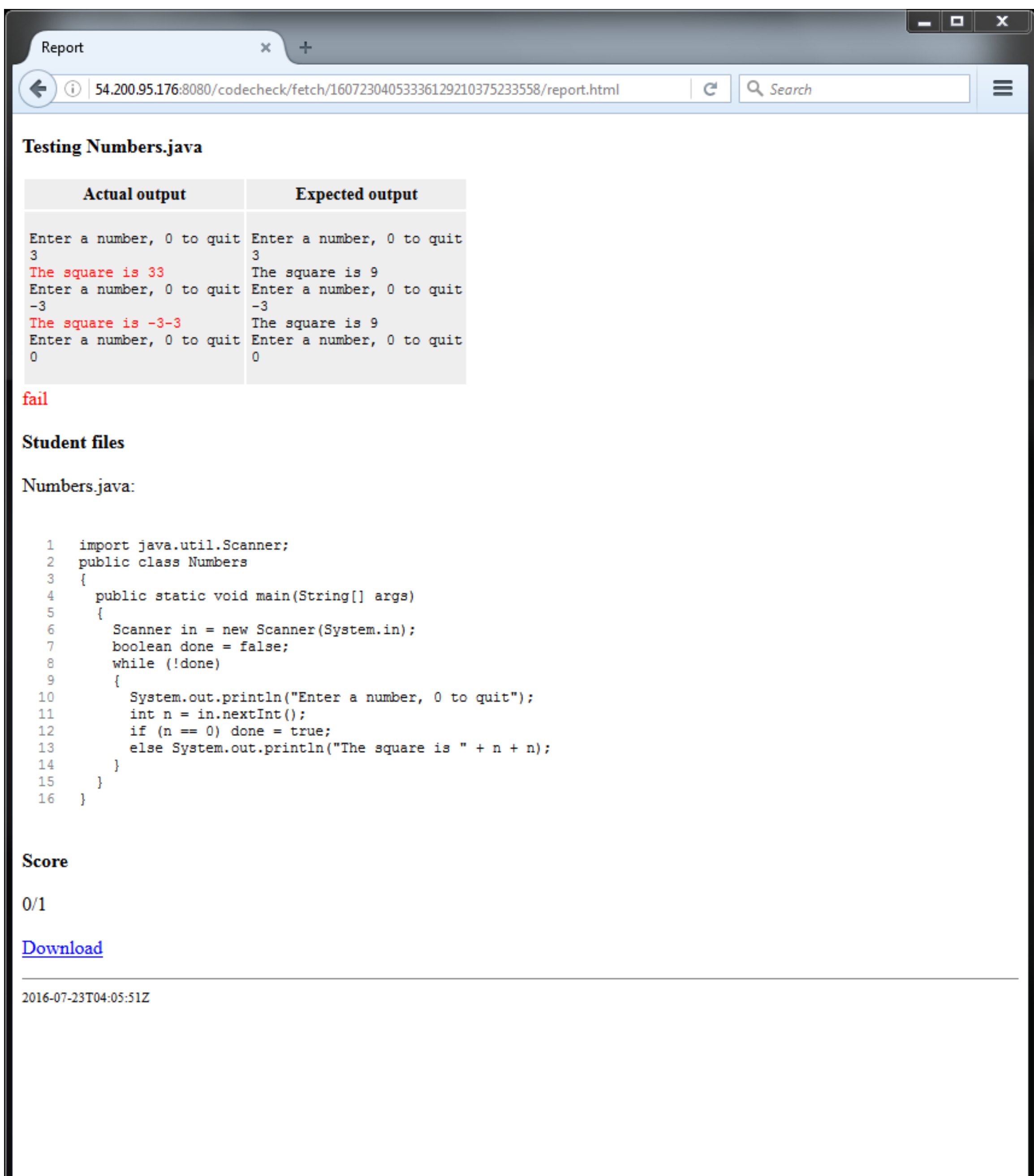


Figure 9: CodeCheck - Report with test failing for the simple problem

Upon a successful test, the 'pass' validation report is given. As shown in Figure 10, there is a 'Download' link at the bottom to save the report. Students can use

this downloaded file as proof of completing the assignment.

Figure 10: CodeCheck - Report with test passing for the multi-file problem

3.2 Web-based code editors

3.2.1 Ace

Ace is an embedded code editor written in JavaScript [1]. Its scripts dynamically modify HTML `div` elements and turn them into IDE-like editors. Ace supports several functions similar to an IDE such as syntax highlighting, bracket matching, auto-completion, auto-indentation and many others. Unlike CodeMirror, Ace has direct support for the Java language.

To add Ace into a HTML page, the desired addon scripts need to be loaded. Each HTML `div` needs to be modified by calling the `ace.edit()` function. An example of Ace is shown in Listing B.1 and Figure 11.

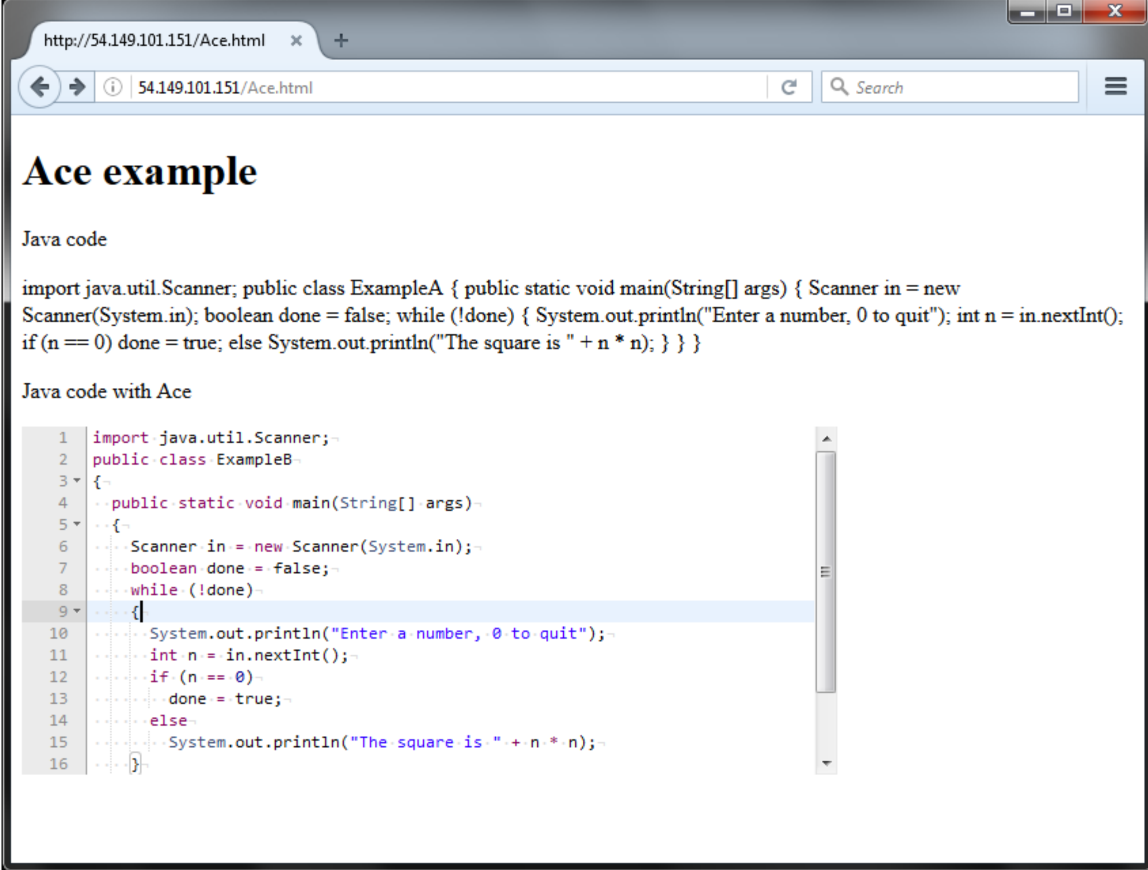


Figure 11: Ace example

In this example, there are two identical `div` elements with IDs 'ExampleA' and 'ExampleB'. The latter is modified with the `ace.edit()` function on line 59. The modification includes setting the theme, the language and various options. Ace has all the options that CodeMirror provides and there are several others that are not available in CodeMirror. These include the ability to show the indentation grid (`displayIndentGuides`), to show hidden characters (`showInvisibles`) and to perform syntax checking(`useWorker`).

3.2.2 CodeMirror

CodeMirror is a web-based text editor implemented in Javascript [2]. Its addon scripts and cascading style sheets (CSS) turn a HTML `textarea` into a visual editor similar to one found in an IDE. CodeMirror supports several functions similar to an IDE such as syntax highlighting, bracket matching, auto-completion, auto-indentation and many others.

To add CodeMirror into a HTML page, the desired addon scripts and CSS need to be loaded. Each HTML `textarea` needs to be modified by calling the `CodeMirror.fromTextArea()` function. An example of CodeMirror is shown in Listing B.2 and Figure 12.

In this example, there are two identical `textarea` elements with IDs 'ExampleA' and 'ExampleB'. The latter is modified with the `CodeMirror.fromTextArea()` function on line 57. The modification includes the display of the line numbers, the auto closing of brackets, the auto code completion and the identification of matching brackets. In Figure 12, the brackets on line 9 and 16 are shown to be matched by switching their font's color to green.

The Java keywords are not highlighted properly because CodeMirror does not

http://54.149.1...odeMirror.html

54.149.101.151/CodeMirror.html

Search

CodeMirror example

Java code

```
import java.util.Scanner;
public class ExampleA
{
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);
    boolean done = false;
    while (!done)
    {
      System.out.println("Enter a number, 0 to quit");
      int n = in.nextInt();
      if (n == 0)
        done = true;
      else
        System.out.println("The square is " + n * n);
    }
  }
}
```

Java code with CodeMirror

```
1  import java.util.Scanner;
2  public class ExampleB
3  {
4    public static void main(String[] args)
5    {
6      Scanner in = new Scanner(System.in);
7      boolean done = false;
8      while (!done)
9      {
10       System.out.println("Enter a number, 0 to quit");
11       int n = in.nextInt();
12       if (n == 0)
13         done = true;
14       else
15         System.out.println("The square is " + n * n);
16     }
```

Figure 12: CodeMirror example

have specific Java language support. This also prevents the automatic code completion from working properly.

3.3 CloudCoder

CloudCoder is a web-based programming exercise system [10]. It provides a platform where users can practice their programming skills and where instructors can assess their students' progress. The system supports multiple classes and programming exercises in C/C++, Python, Java and Ruby.

As shown in Figure 13, the CloudCoder system is composed of two components: the web application (webapp) and the builder. The webapp manages the user interfaces and the communication with the database and the builder. In the current deployment, MySQL is used to store information on the users, the classes, the exercises and the state of users. The builder processes users' submissions and executes the compiled programs against a predefined set of test cases.

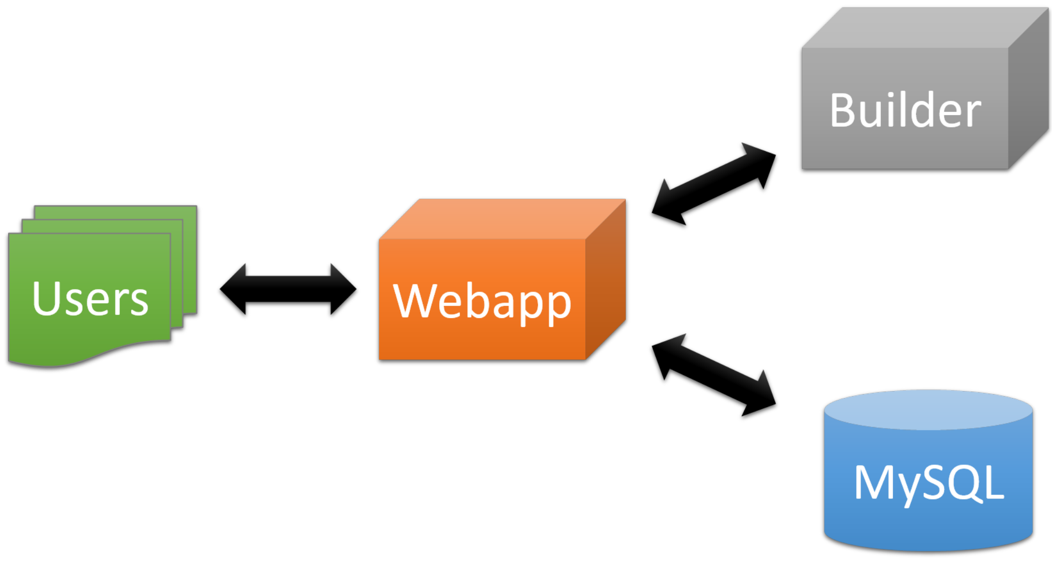


Figure 13: CloudCoder - System

Students and instructors interact with CloudCoder through a browser. CloudCoder supports basic authentication and users have access to only the classes they are assigned to. The login page is shown in Figure 14. Once logged in, the users

will be given a snapshot of their progress and the list of exercises that they need to complete. An example is shown in Figure 15.

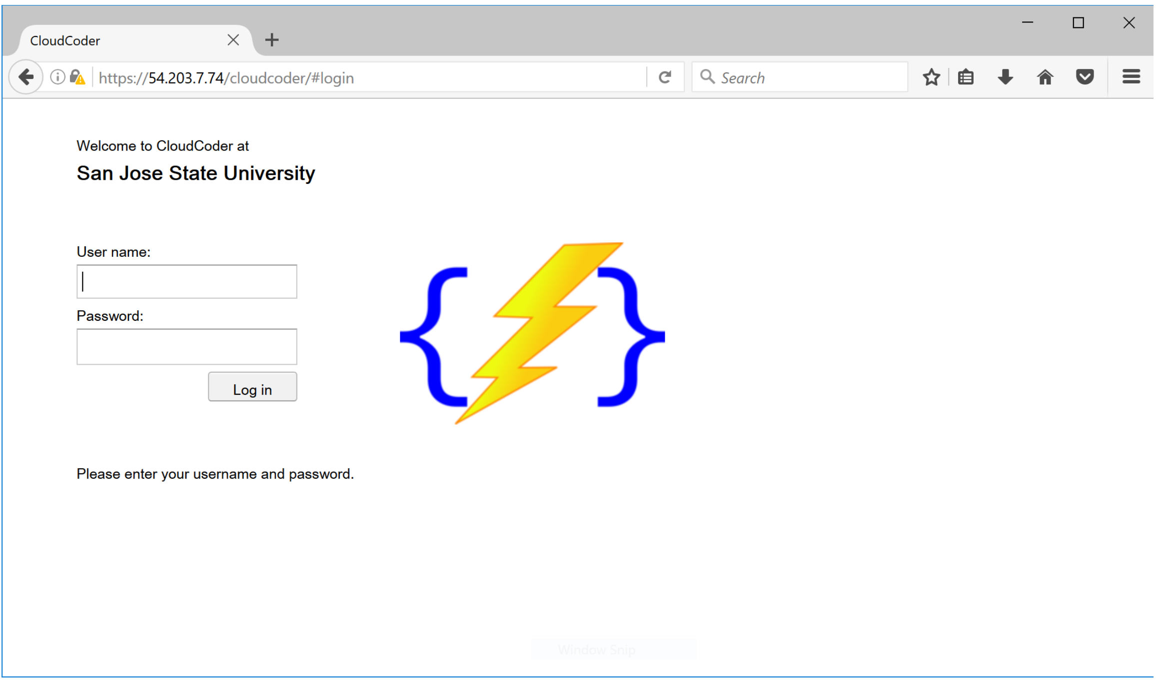


Figure 14: CloudCoder - Login page

With an exercise is loaded, a user can start working on coding the solution in the editor. Submissions are checked by the Builder each time the Submit button is clicked. Figure 16 shows the C "hello world" program. The test failed in this case because of the missing comma in the output string. The state of the code is also saved with each submission. The user can leave the problem via either the Back or Logout button. Upon returning to a problem, the last saved state will be restored so that the user can continue the exercise.

Instructor accounts have two administrative buttons that manage the exercises and the students assigned to the selected class, as shown in Figure 17.

Figure 18 shows the user management page for a class where an instructor can add, edit or delete an individual student's accounts. A tab separated text file with

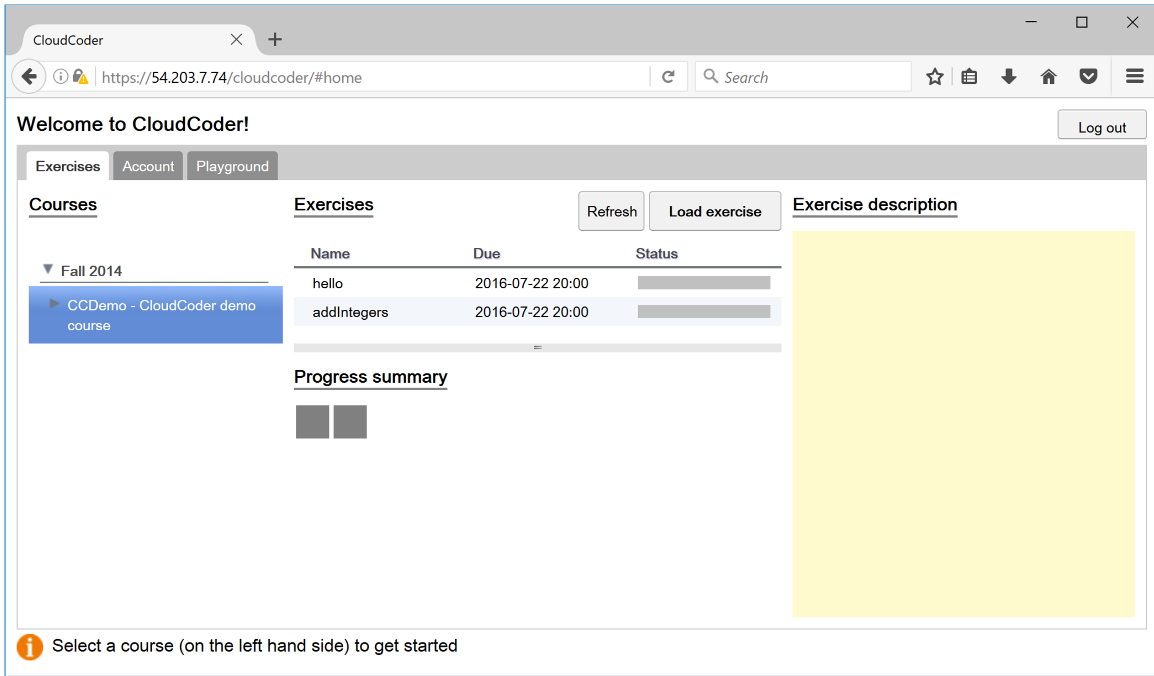


Figure 15: CloudCoder - Student main page

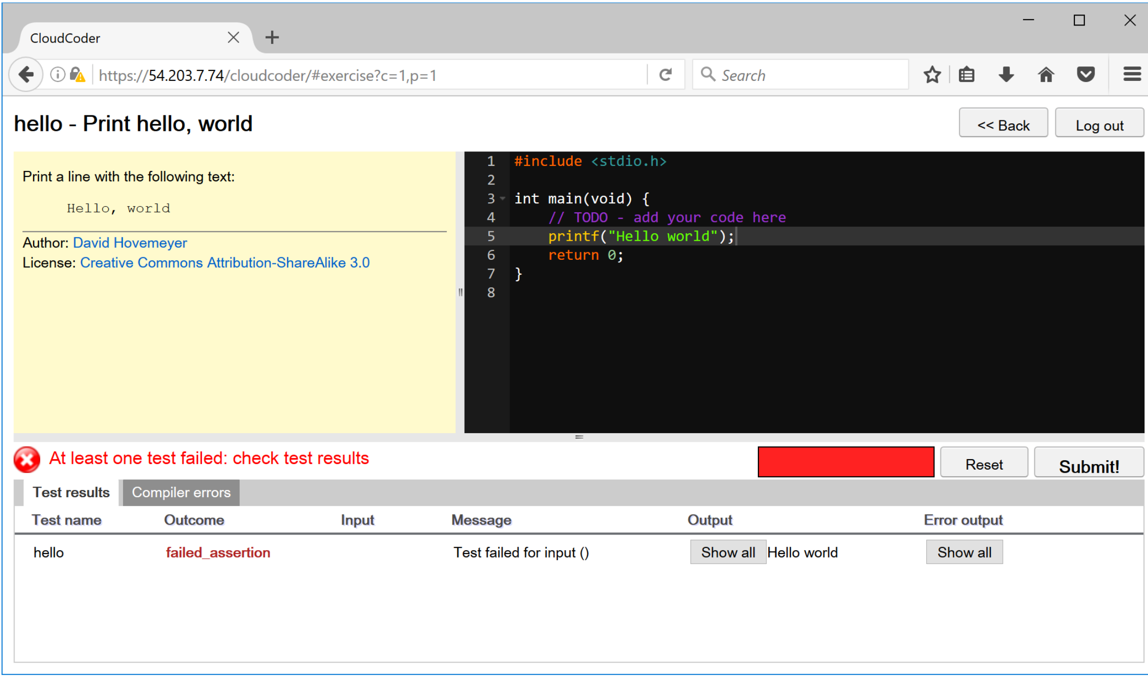


Figure 16: CloudCoder - Student problem page

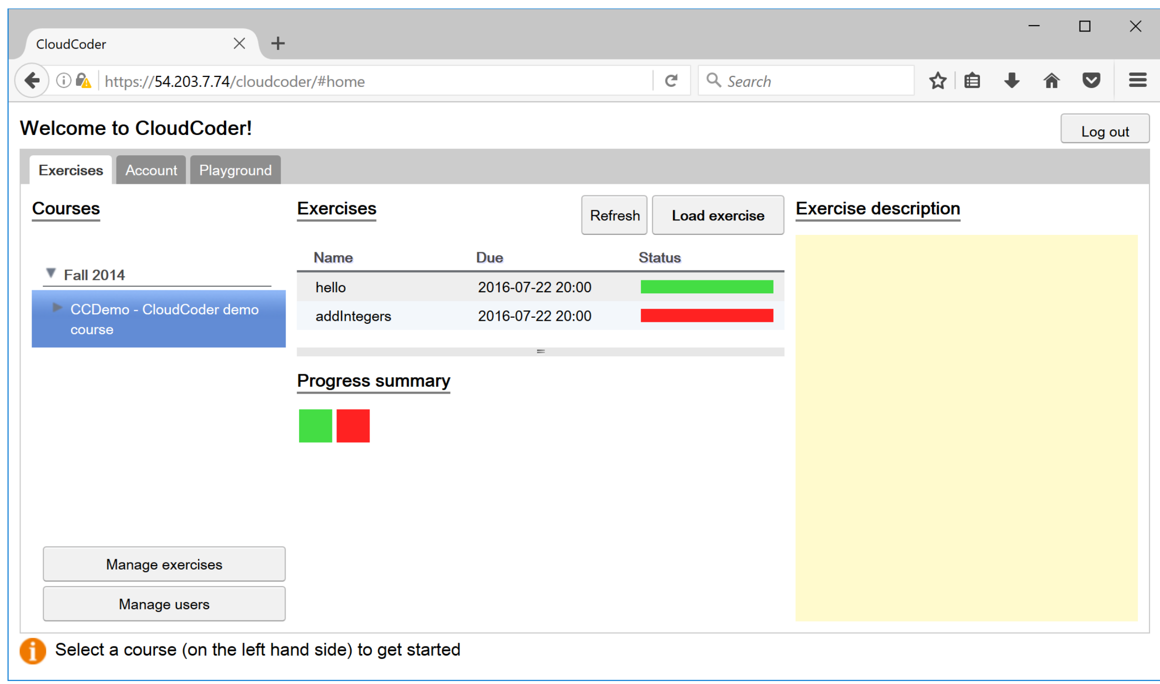


Figure 17: CloudCoder - Instructor main page

a list of students can also be uploaded via the 'Bulk register' button. A student's progress on the exercises can be monitored by selecting the student's name and then clicking on the 'Statistics' button. CloudCoder also allows the administrator to upload a comma separated list of students in a class via the command line.

Exercises can be added to a class via the exercise management page, as shown in Figure 19. Unlike the user management page, exercises cannot be loaded via the command line. However, exercises can be imported from another class ('Import course') or from the exercise repository ('Import'). Various CloudCoder sites can contribute their exercises to the repository by enabling the permissive license ('Make permissive') and by selecting the 'Share' button. The due dates and the publication of the exercises can be set quickly via the 'Set dates/times' and 'Make visible' buttons, respectively.

Figures 20 through 25 show the "Edit problem" page and the various properties

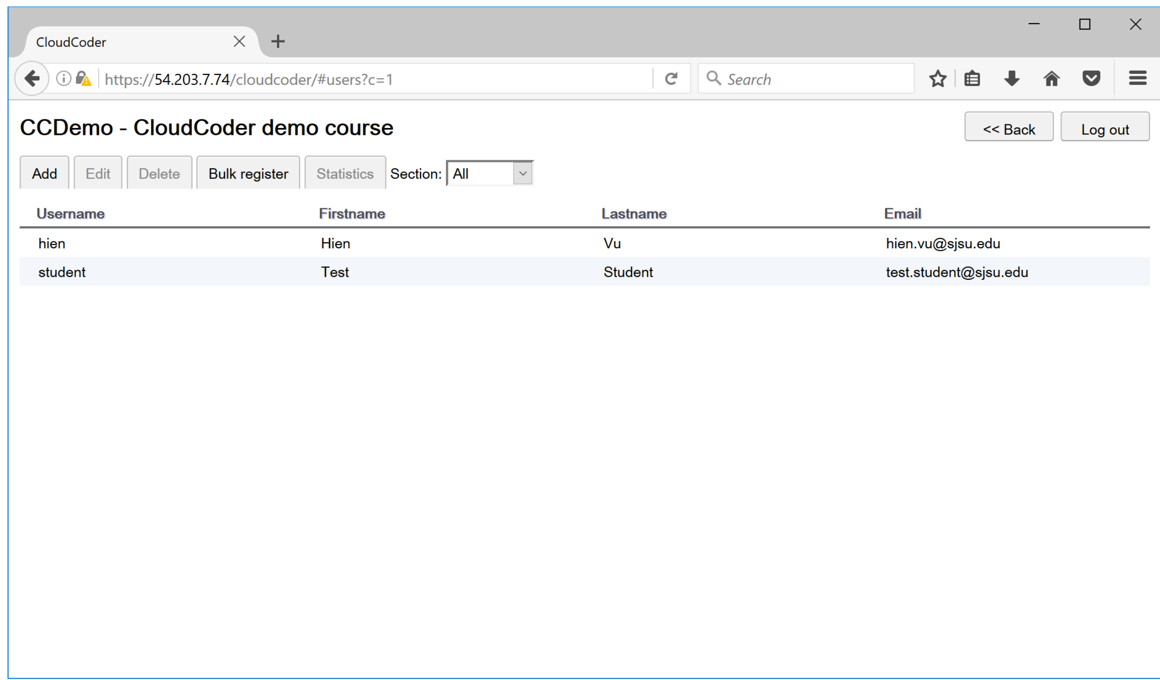


Figure 18: CloudCoder - User management page

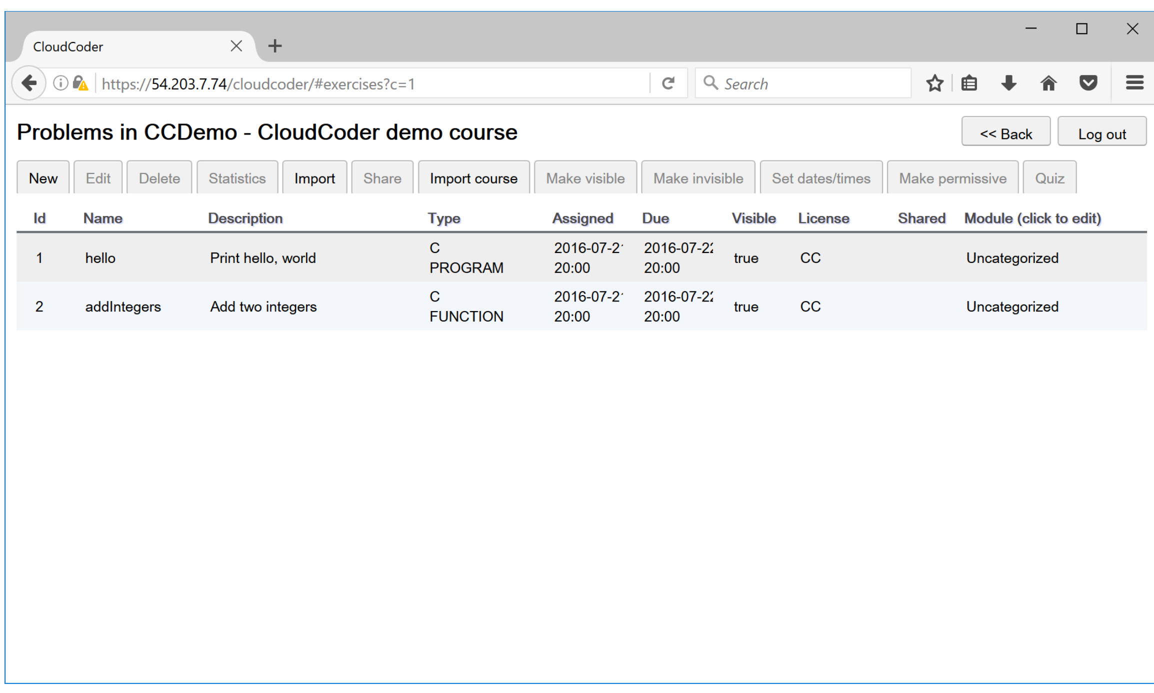


Figure 19: CloudCoder - Exercises management page

of an exercise in CloudCoder. There are 8 supported problem types: Java, C, C++, Python and Ruby functions or methods and Java, C and C++ programs. The full description and skeleton editors are Ace editors. The test cases are regular expressions that will be used to validate the user solution's output. Multiple test cases can be added by clicking on the 'Add Test Case' button.

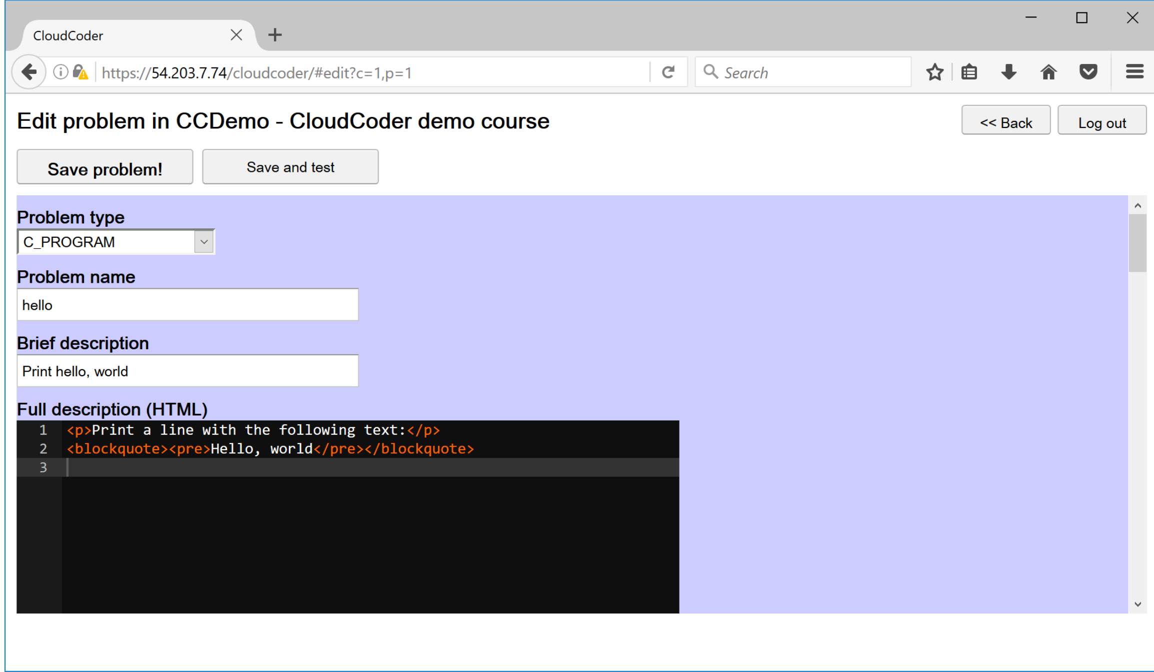


Figure 20: CloudCoder - Exercise composer (1 of 6)

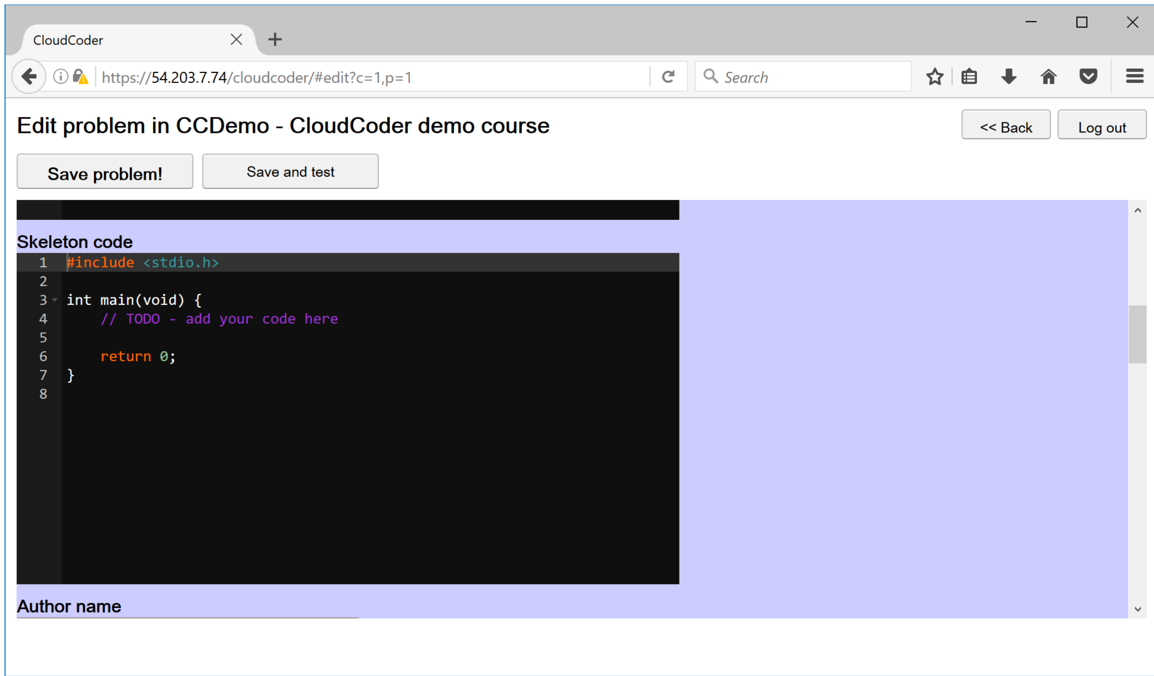


Figure 21: CloudCoder - Exercise composer (2 of 6)

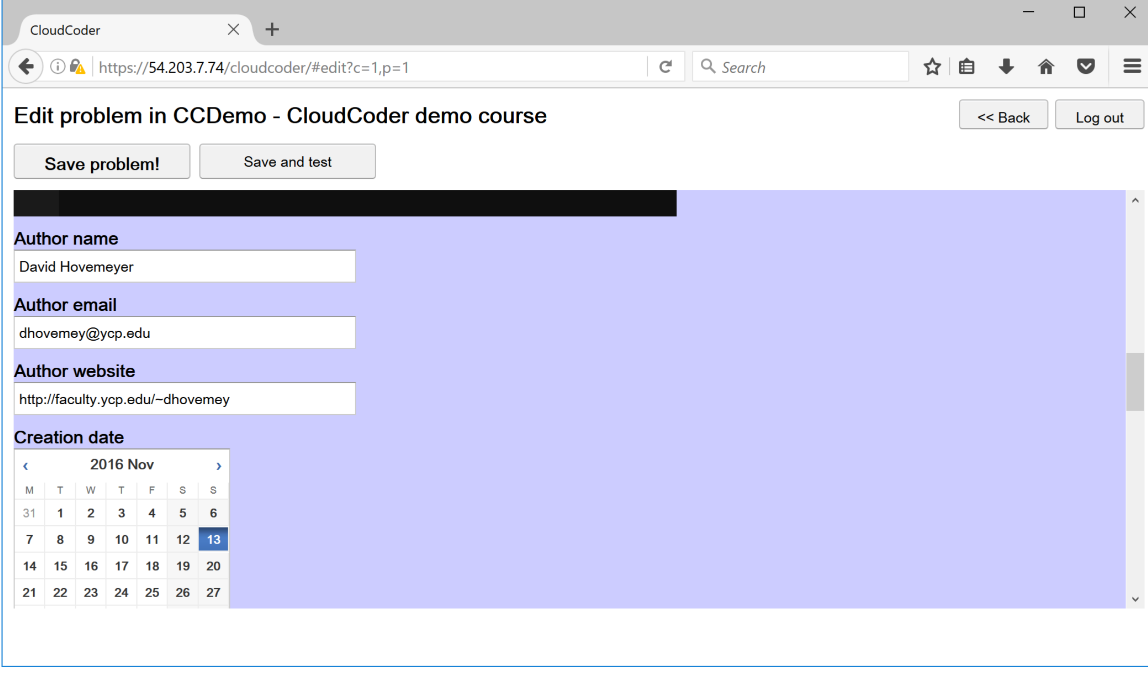


Figure 22: CloudCoder - Exercise composer (3 of 6)

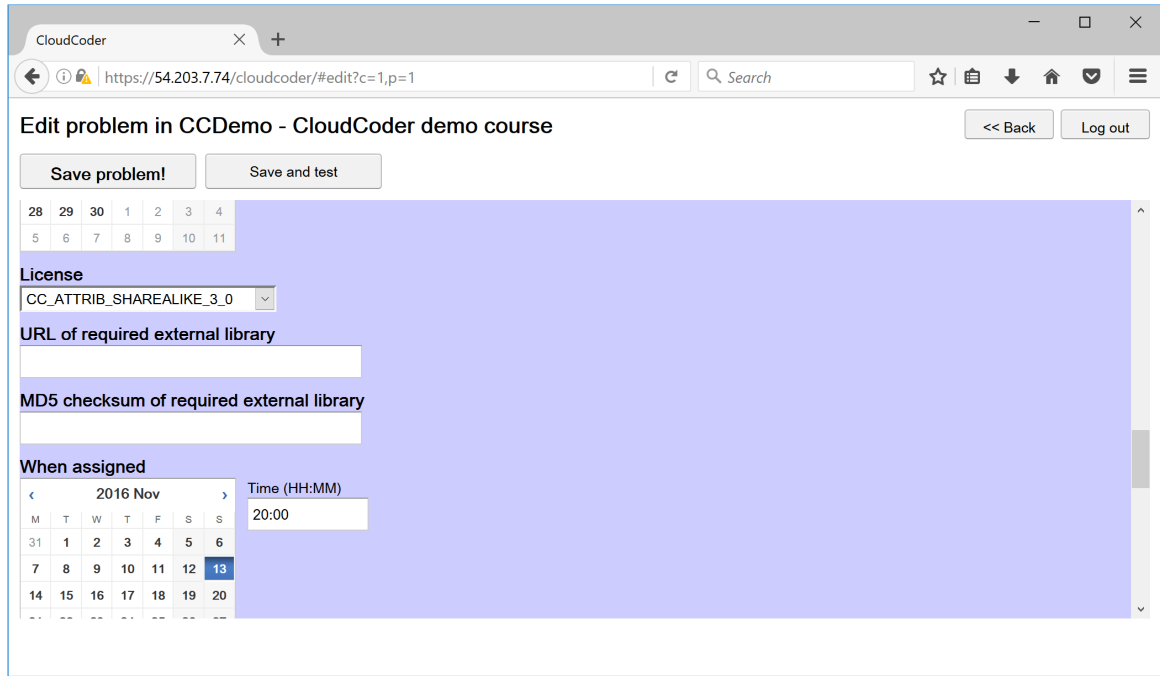


Figure 23: CloudCoder - Exercise composer (4 of 6)

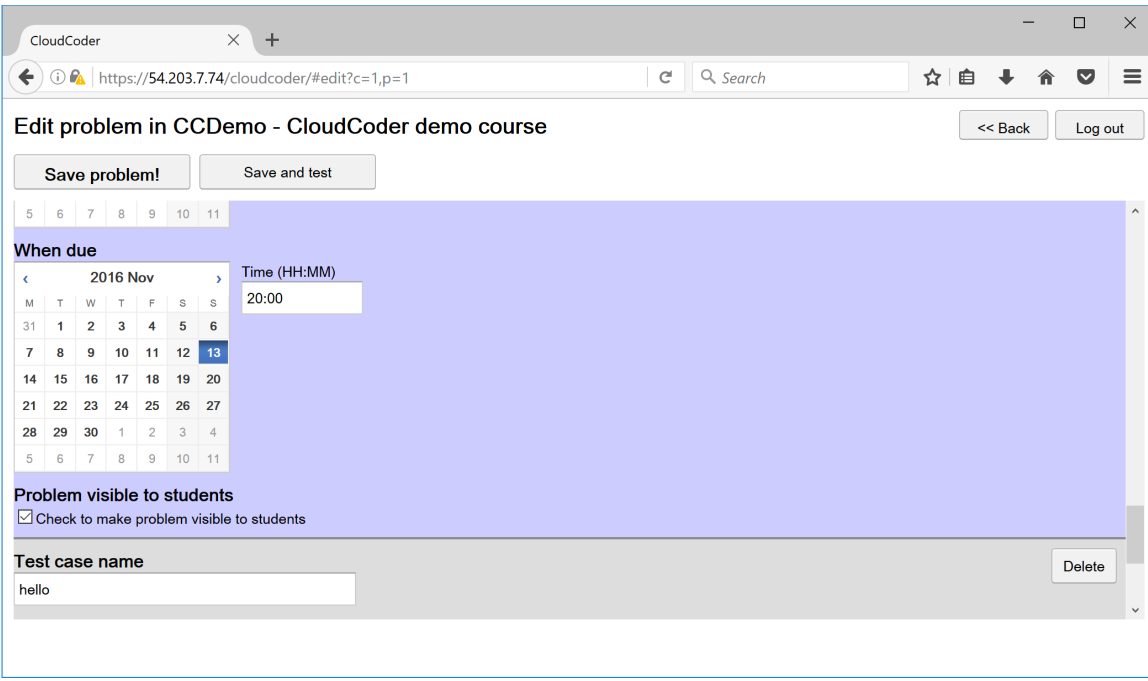


Figure 24: CloudCoder - Exercise composer (5 of 6)

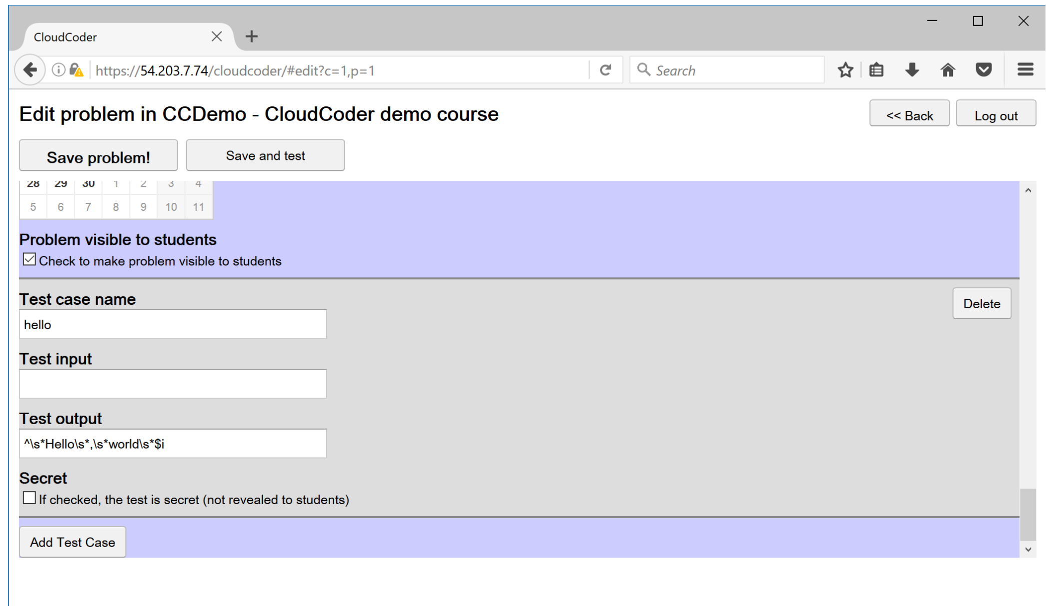


Figure 25: CloudCoder - Exercise composer (6 of 6)

CHAPTER 4

Integration of Ace and CodeMirror with CodeCheck

As indicated earlier, Codecheck does not allow the instructor to update a problem directly. When an issue is encountered, a new zip file has to be uploaded. One of the first enhancements done in this project is to provide an instructor interface. With a successful upload of the problem, the new page containing the randomly generated URL also has a link to 'Update' the problem. This is shown in Figure 26.

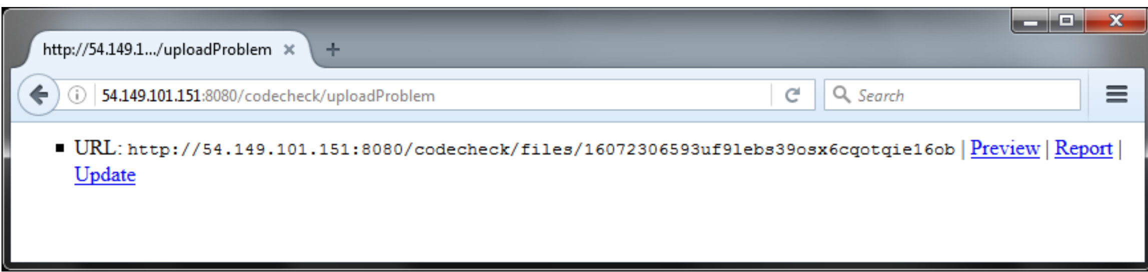


Figure 26: Codecheck - Enhanced uploaded problem page

The 'Update' page allows the instructor to modify the existing files as well as adding new one. An example of this is shown in Figures 27 and 28.

The enhanced starting web page of the problem is shown in Figure 29.

Both instructor and student pages were enhanced with Ace and CodeMirror editors respectively. JQuery functions were created to simplify the HTML reformatting. Listing 4.1 shows an example JQuery function that modified all textarea by referencing their IDs.

```
1 $(document).ready(function() {
2   $("textarea").each(function(index) {
3     var textarea_id = $(this).attr('id');
```

Figure 27: Codecheck - Instructor interface for a simple problem

```
4        var editor = CodeMirror.fromTextArea(document.getElementById(
           ↪ textarea_id), {
5          extraKeys: {"Ctrl-Space": "autocomplete"},
6          lineNumbers: true,
7          viewportMargin: Infinity,
8          matchBrackets: true,
9          styleActiveLine: true,
10         autoCloseBrackets: true,
11         showTrailingSpace: true,
```

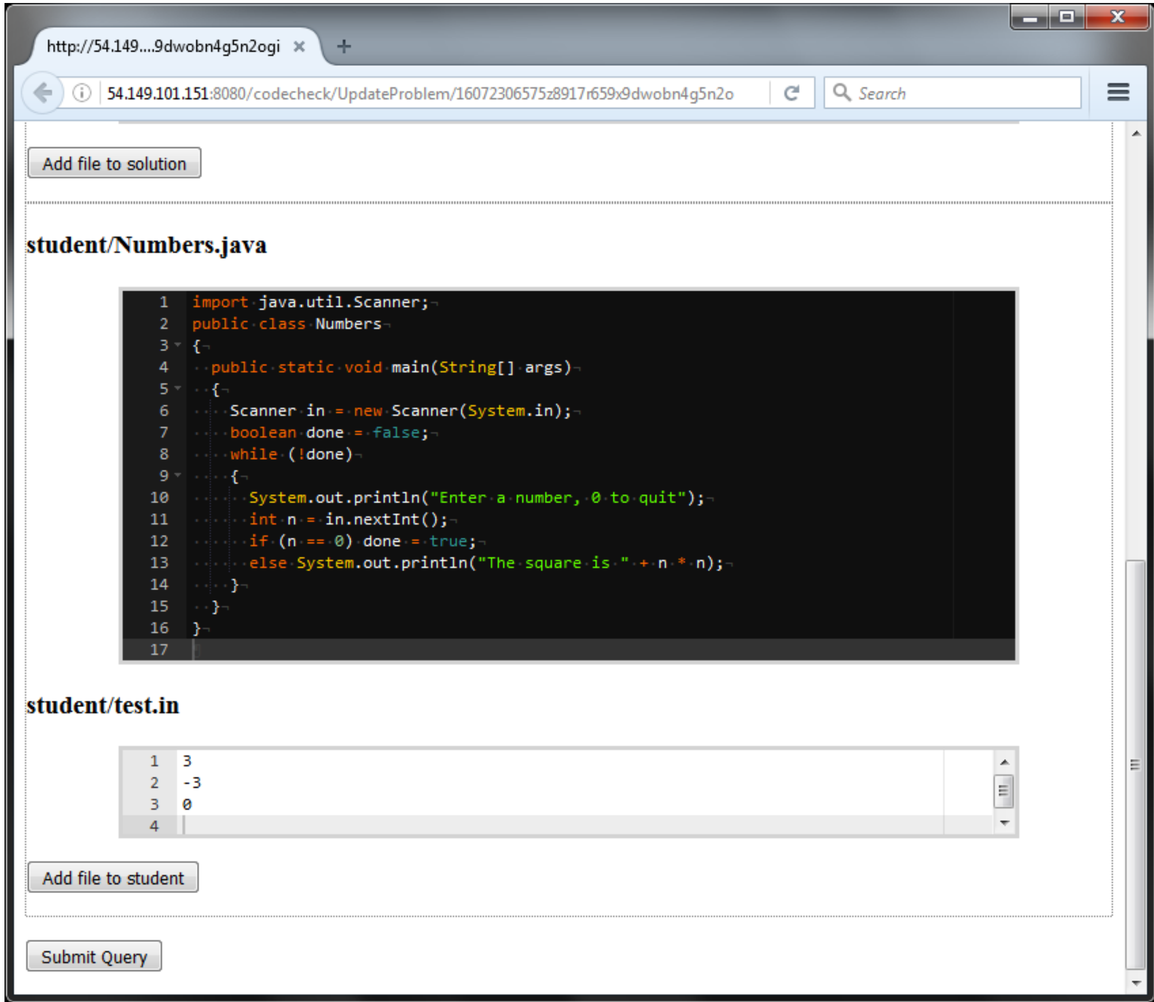


Figure 28: Codecheck - Instructor interface for a simple problem

```
12        theme: 'neo',
13        tabSize: 2
14      });
15    });
16  });
```

Listing 4.1: CodeMirror formatting

Implementing the support for Ace is greatly simplified by using the jQuery-Ace plugin [15]. Java source files in a HTML page can be classified using the HTML's

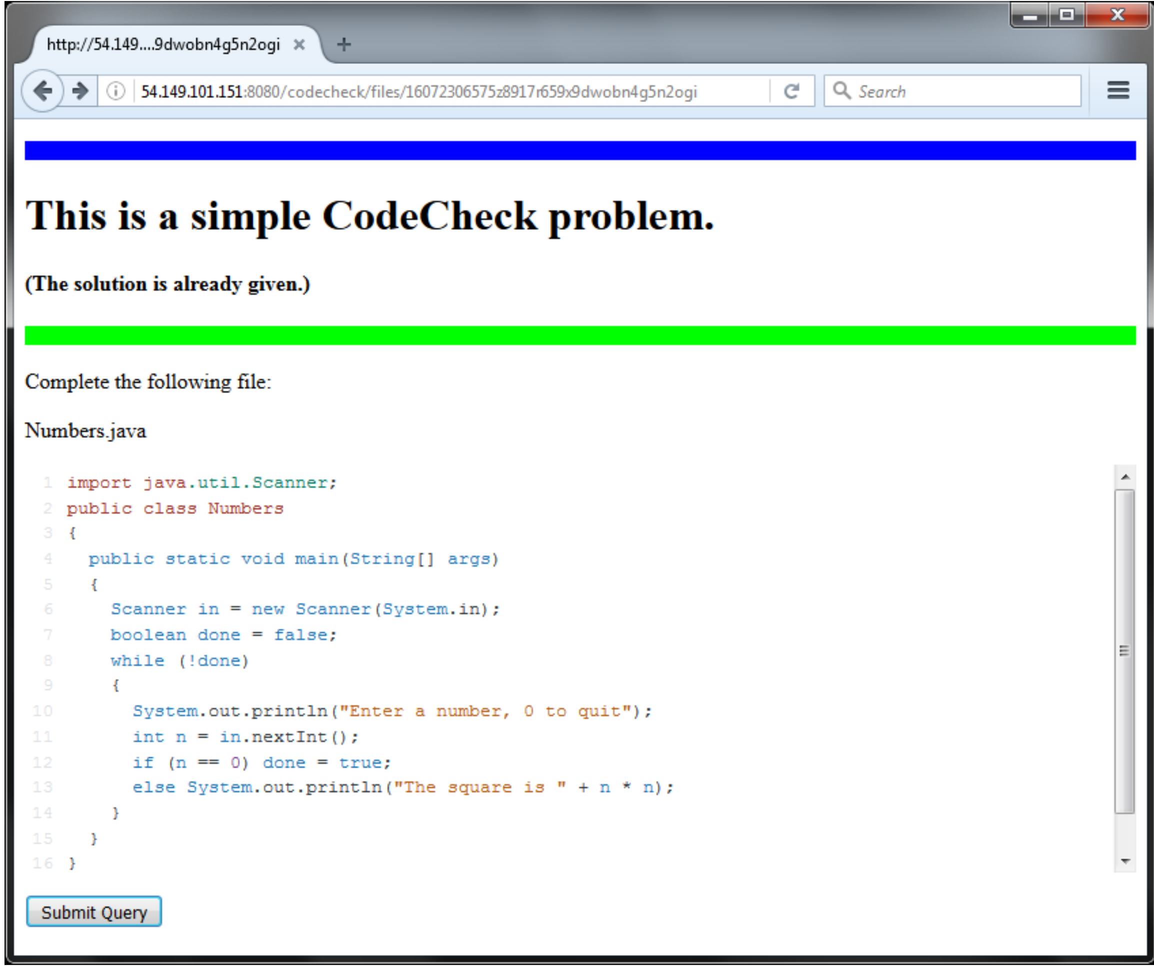


Figure 29: Codecheck - Enhanced student interface for a simple problem

`class` elements and the jQuery code in Listing 4.2 will modify them accordingly.

```
1  $('.java').ace({ theme: 'vibrant_ink', lang: 'java' }).each(function
     ↪ (idx, editor) {
2    var ace = $(editor).data('ace').editor.ace;
3    ace.setOption("autoScrollEditorIntoView", "true");
4    ace.setOption("displayIndentGuides", "true");
5    ace.setOption("enableBasicAutocompletion", "true");
6    ace.setOption("enableLiveAutocompletion", "false");
7    ace.setOption("enableSnippets", "true");
8    ace.setOption("maxLines", 30);
```

```
9     ace.setOption("showInvisibles", "true");
10    ace.setOption("tabSize", 2);
11    ace.setOption("useWorker", "true");
12  });
13  $('.cpp').ace({ theme: 'tomorrow_night', lang: 'c_cpp' }).each(
      ↪ function(idx, editor) {
14    var ace = $(editor).data('ace').editor.ace;
15    ace.setOption("autoScrollEditorIntoView", "true");
16    ace.setOption("displayIndentGuides", "true");
17    ace.setOption("enableBasicAutocompletion", "true");
18    ace.setOption("enableLiveAutocompletion", "false");
19    ace.setOption("enableSnippets", "true");
20    ace.setOption("maxLines", 30);
21    ace.setOption("showInvisibles", "true");
22    ace.setOption("tabSize", 2);
23    ace.setOption("useWorker", "true");
24  });
25  $('.text').ace({ theme: 'terminal', lang: 'plain_text' });
26  $('.html').ace({ theme: 'chrome', lang: 'html' });
27  $('.default').ace({ theme: 'xcode', lang: 'text' });
```

Listing 4.2: Ace formatting

With the integration of the Ace/CodeMirror editor and the addition of the instructor's interface, the usability of CodeCheck was improved. However, it is still not possible to track the student's progress. Also, the student cannot resume working on an unfinished problem.

CHAPTER 5

Integration of CloudCoder with CodeCheck

The integration of CloudCoder and CodeCheck consists of the replacement of the builder and the support for CodeCheck exercises. The enhanced CloudCoder system is shown in Figure 30. The swap of the builder with CodeCheck required a change in the way exercise's submissions and their results are transferred between the webapp and the builder. The JavaScript Object Notation (JSON) format was chosen for its simplicity; more details can be found in section 5.3. The added support for CodeCheck exercises is more complex because several sub-components of CloudCoder had to be changed. CloudCoder uses the Model-View-Controller (MVC) architecture so the *model* and the *view* components of this framework have to be enhanced. Sections 5.1 and 5.2 discuss in details the changes in the data model and the user interfaces.

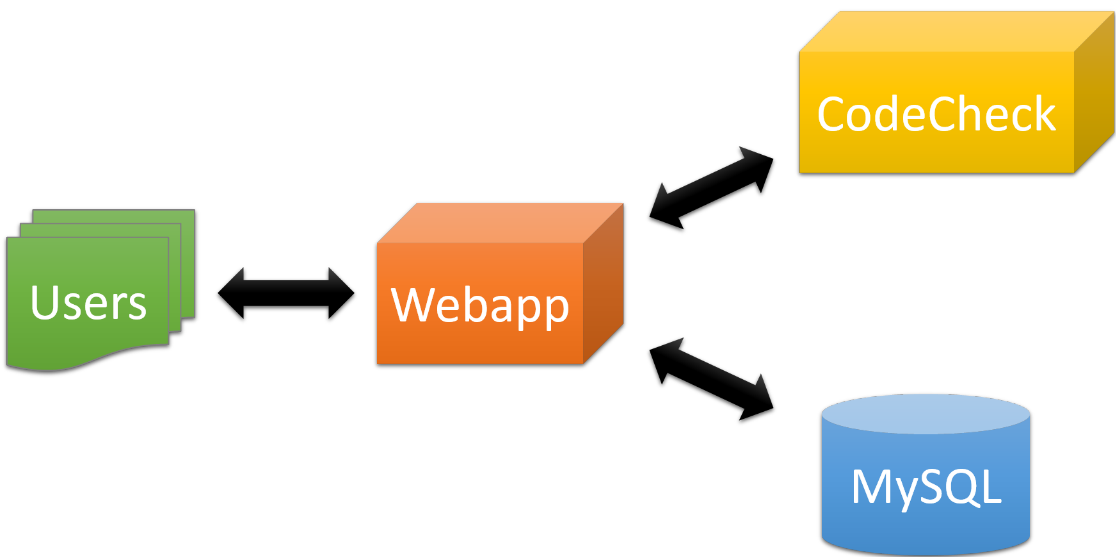


Figure 30: CodeCheck - Enhanced system

5.1 Data model

One of the main differences between a CloudCoder exercise and a CodeCheck exercise is the number of user files and how the users' solutions are validated. CloudCoder provides a single *skeleton* file to a user and it validates the user's submissions by checking the submitted code's output using regular expressions. CodeCheck's exercise can have one or more *skeleton* files and it also requires the same number of corresponding solution files in order to validate the user's submissions. To support CodeCheck's exercises, CloudCoder was enhanced to support multiple *student* and *solution* files.

Data related to a CloudCoder exercise is modeled by the `Problem` class. The test case is modeled separately by the `Testcase` class. These two objects are combined to form the `ProblemAndTestCaseList` object. The remote procedure calls (RPC) of the Google Web Toolkit (GWT) were implemented to transfer these data between the various components of CloudCoder.

The *student* and *solution* files are modeled as objects with two main attributes: name and content. The other attributes are necessary for the storing of these objects in the database. To leverage the existing RPC implementation, the `ProblemAndTestCaseList` class was enhanced to additionally contain lists of `StudentDirVTH` and `SolutionDirVTH` objects.

All persistent data are stored in a MySQL database. So several JDBC database transactions have been implemented. The *solution* files are retrieved by the client's UI so special transactions were implemented to return a `Base64` encoded string. The `CreateWebappDatabase` class was enhanced to include the `StudentDirVTH`'s and `SolutionDirVTH`'s schema. This class is used to setup the database inside MySQL.

The `StoreProblemAndTestCaseList` transaction class was enhanced to also write the `student` and `solution` files to the database.

Two editors classes were implemented to support the *student* and *solution* files. The Ace editor is part of CloudCoder so features such as syntax highlighting, auto-indentation and auto-completion were enabled to enhance the users' experience. The `StudentDirDataEditor` and `SolutionDirDataEditor` classes also implemented a "Delete" button to enable the removal of a *student* or *solution* file and its corresponding editor from the UI.

5.2 Views

5.2.1 Edit problem page

The "Edit problem" page is composed of panels and widgets from the GWT library. To support the creating and editing of the *student* and *solution* files, two GWT `FlowPanel` panels were added to the page. Initially the panels contain only the "Add Student File" and the "Add Solution File" buttons. The `Buttons`' `addClickHandler()` method instantiates a `StudentDirVTH` or `SolutionDirVTH` object and adds it to the `ProblemAndTestCaseList` object residing in the client's session. An instance of the `StudentDirDataEditor` or `SolutionDirDataEditor` class is also instantiated and is added to panel.

Most of the code involving the client/server data transfer does not need to be modified because the *student* and *solution* files are part of the `ProblemAndTestCaseList` object. However, one method needed to be enhanced because it retrieves only the problem's `TestCase` using the `Problem` object. The `loadProblemAndTestCaseList()` method is a static method in the `SessionUtil` class and is called by the `ProblemAdminPage` class, which handles the "exercises

management" page. The `TestCase` is retrieved via an RPC call to the server. So in order to also retrieved the `student` and `solution` files via their respective RPC calls, the RPC calls are chained together as described by Northrop [17]. The `student` and `solution` RPC calls are described in Section 5.2.2.

To reduce the number of required fields, the problem's type was removed and all problems have the default "CODECHECK TYPE" type. The "Skeleton code" editor was removed and was replaced by the *student* and *solution* panels. The "License" field was removed because these CodeCheck exercises are not shareable to other CloudCoder servers. The "URL of required external library" and "MD5 checksum of required external library" were removed because the CodeCheck problem type does not support external libraries. Also, the testcases for CodeCheck are included as part of the *student* files so the "Test Case" section as shown in Figure 25 was removed.

The enhanced interface is shown in Figures 31 through 36.

5.2.2 Development page

The "Development" page is also composed of panels and widgets from the GWT library. To support multiple *student* files, the `editorLayoutPanel` object's type was changed from a `LayoutPanel` to a `TabLayoutPanel`. This is to support multiple *skeleton* files. Also, to maintain the *student* and the *solution* files, several `ArrayList` instances were created. In the UI's `activate()` method, the `getStudentDir()` and `getSolutionDir()` method calls use RPC to retrieve the *student* and *solution* files from the webapp. To leverage the existing framework, these RPC methods were implemented in the `GetCoursesAndProblemsServiceImpl` class. Because the RPC calls are asynchronous, a fixed number of Ace editors is created up front and stored in

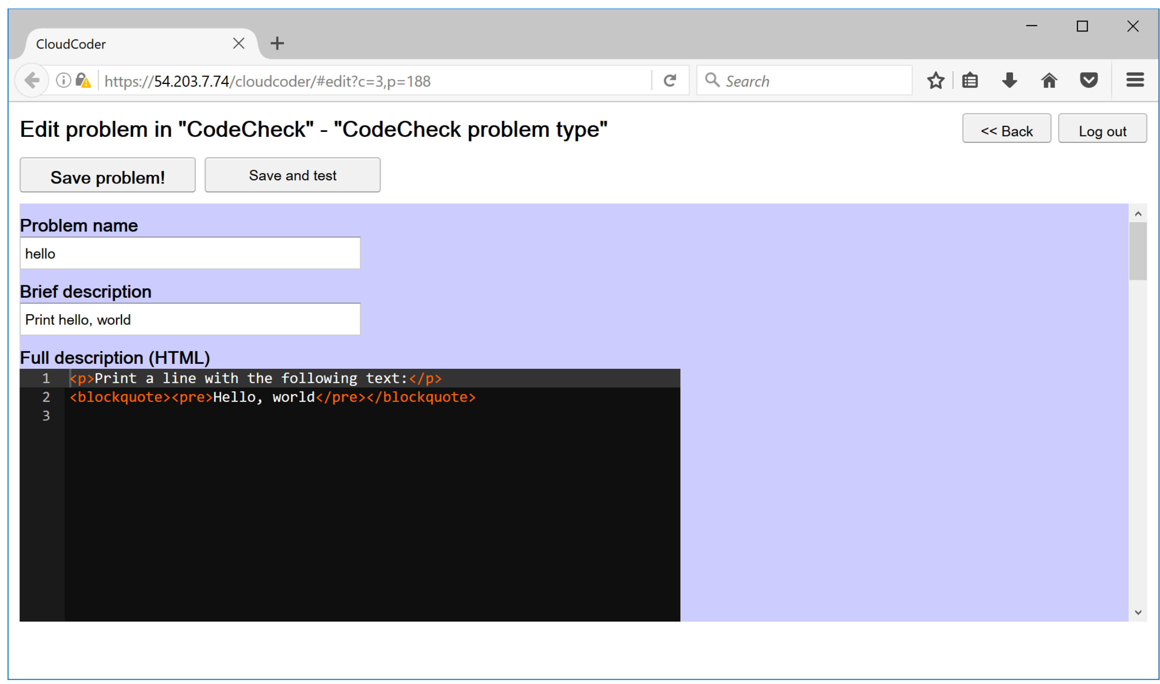


Figure 31: CodeCheck - Enhanced exercise composer (1 of 6)

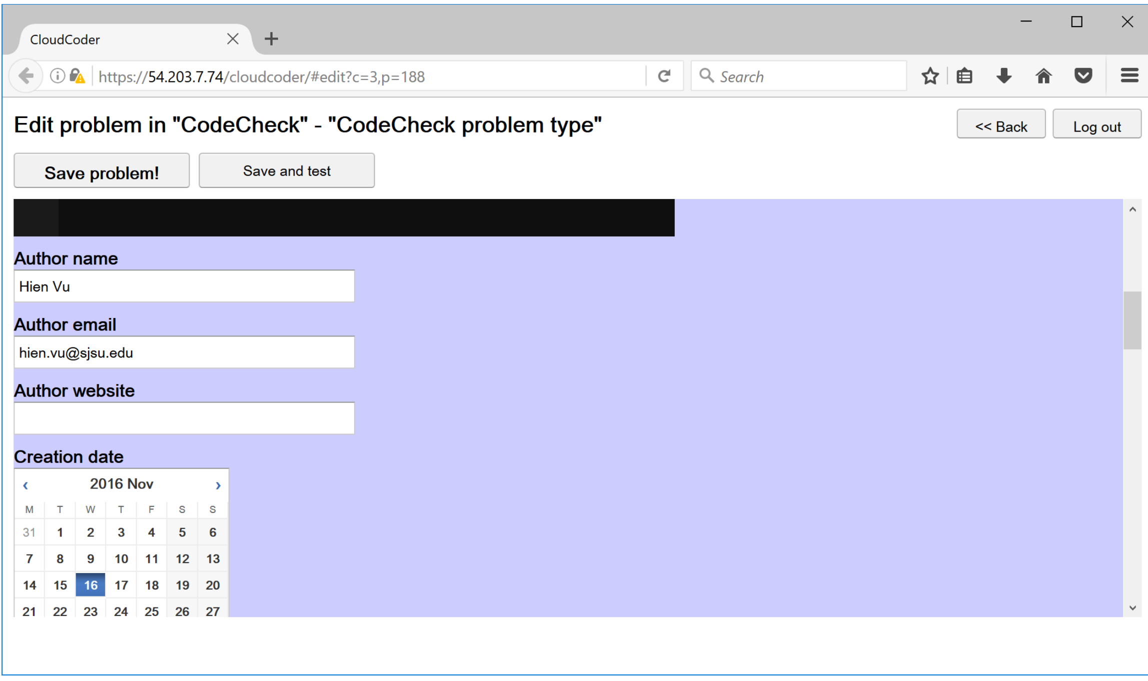


Figure 32: CodeCheck - Enhanced exercise composer (2 of 6)

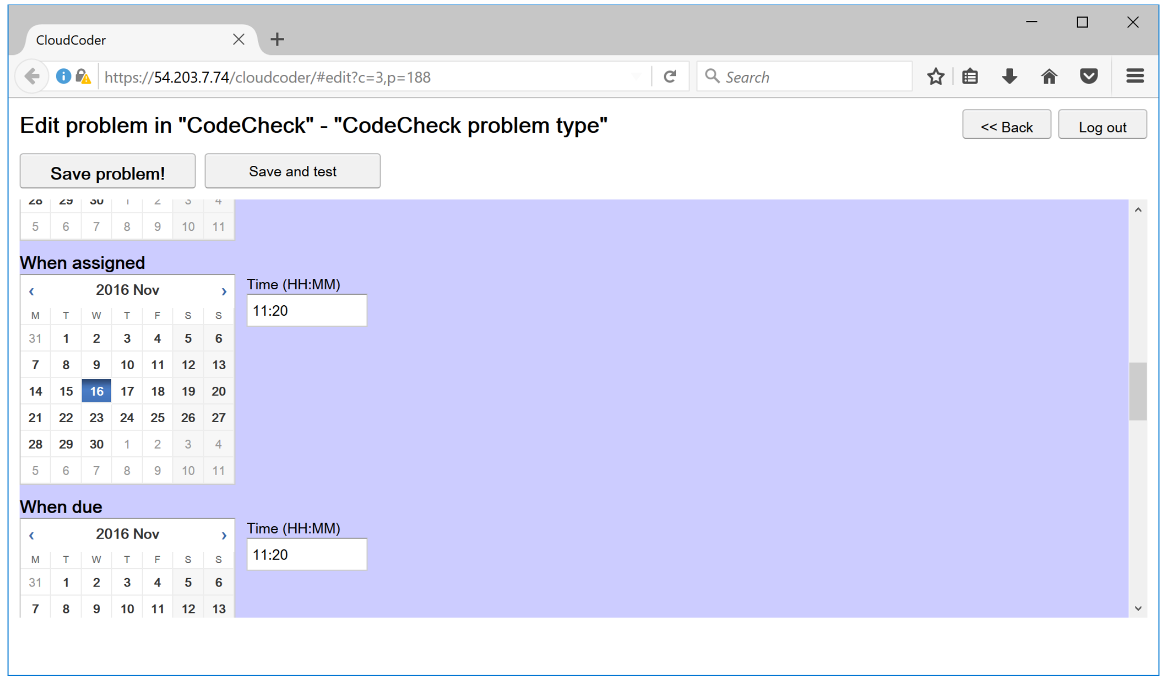


Figure 33: CodeCheck - Enhanced exercise composer (3 of 6)

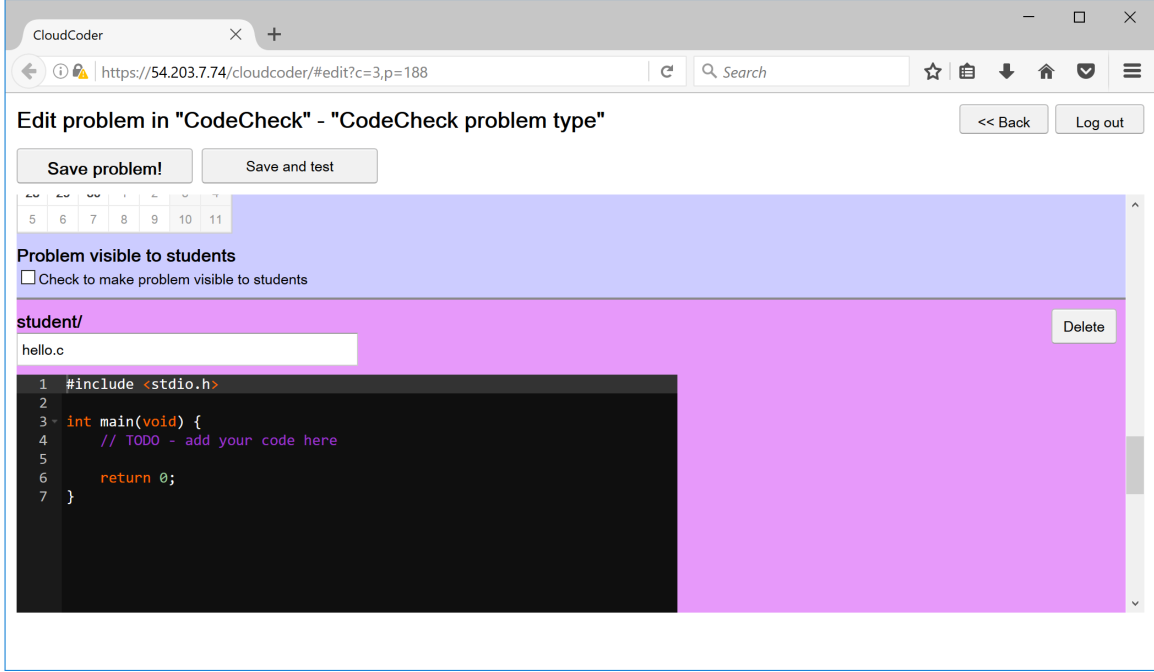


Figure 34: CodeCheck - Enhanced exercise composer (4 of 6)

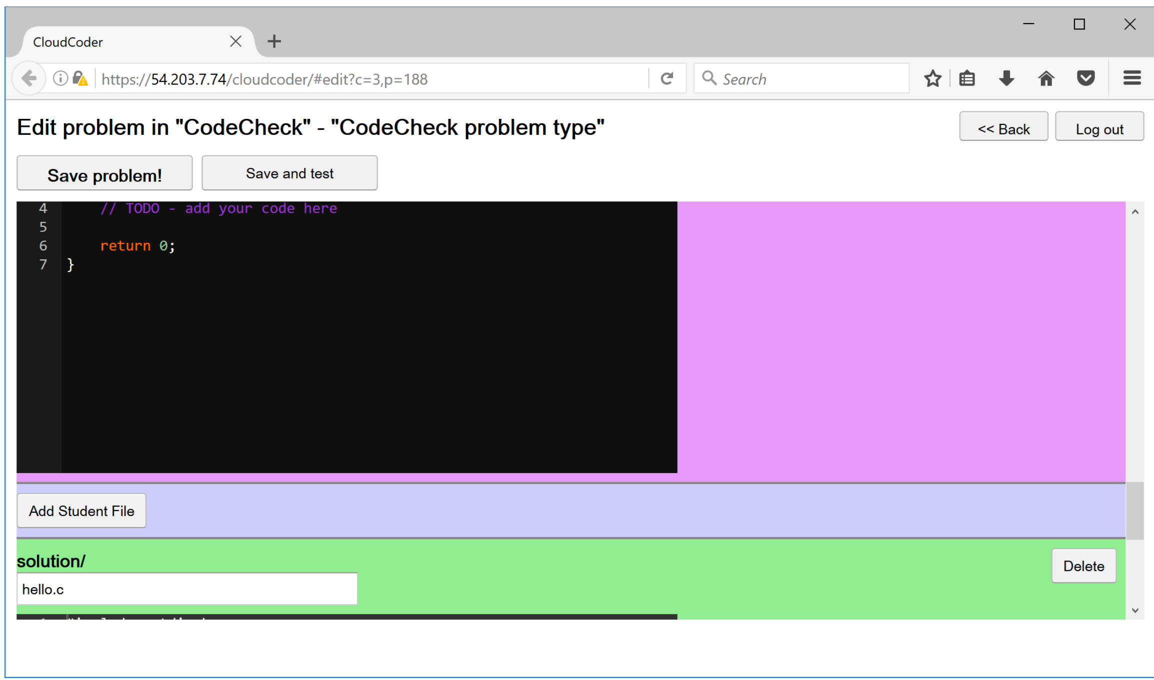


Figure 35: CodeCheck - Enhanced exercise composer (5 of 6)

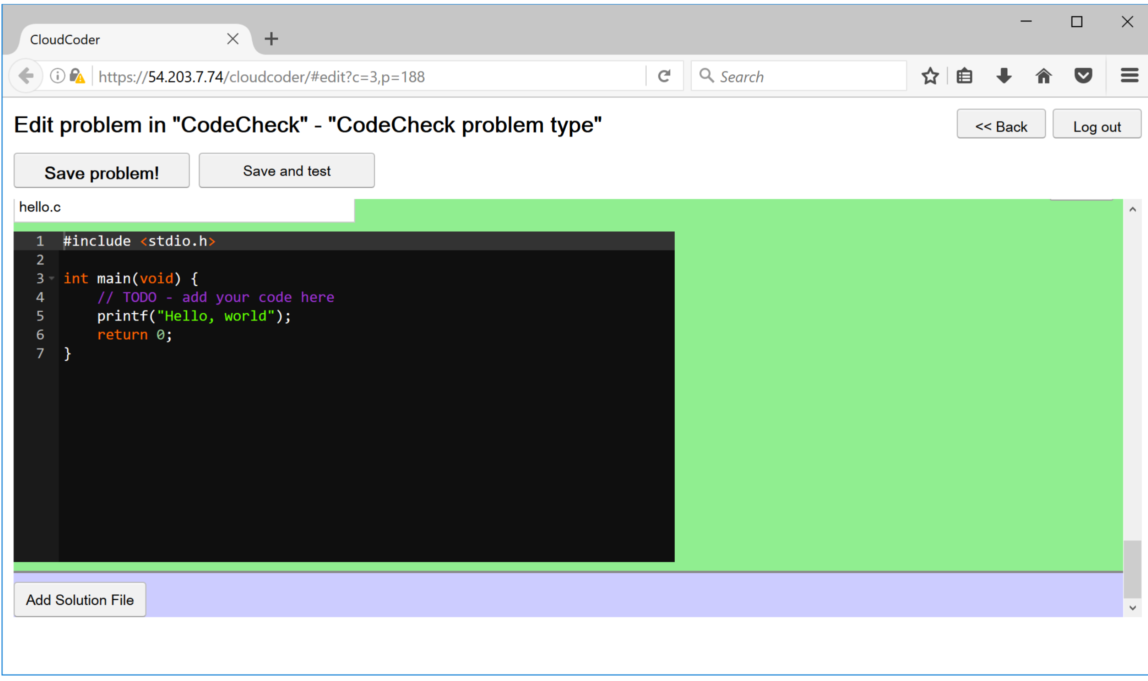


Figure 36: CodeCheck - Enhanced exercise composer (6 of 6)

the `aceEditors` list. The list is trimmed by the `asyncLoadCurrentProblemText()` method when the editors are being initialized with the contents.

The "Submit" button's handler was modified to call the `doCodeCheckSubmit()` method. The method extracts the *student*, the *solution* and the *submission* files, merges them into a JSON string and sends it to the webapp by the `doSubmitCodeCheckRPC()` method. This method triggers the `submitCodeCheck()` RPC call to the server and waits for the submission's result. The `onReceiveSubmissionResult()` method was enhanced to parse the diagnostic message and annotate the corresponding editors when compilation failures occur.

The `submitCodeCheck()` RPC method is implemented in the `SubmitServiceImpl` class. The method saves the editor changes made by the users to the database, instantiates an `IFutureSubmissionResult` object and adds it to the session. The `IFutureSubmissionResult` object is the result of the submission which is asynchronously obtained from the `submitCodeCheckAsync()` method implemented in the `OutOfProcessSubmitService` class. The `submitCodeCheckAsync()` method opens an HTTP connection to the CodeCheck server, sends the JSON and retrieves the returned JSON. Upon receiving the returned JSON, a `OOPCodeCheckSubmission` object, whose class implements the `IFutureSubmissionResult` interface, is created to parse the JSON and populate its `SubmissionResult` attribute.

5.3 Webapp and CodeCheck communication

The communication between CloudCoder and CodeCheck is done via JSON strings. To support this project, CodeCheck was enhanced to support incoming submissions via an HTTP POST request.

The submission string consists of a JSON object with 4 set of name/value pairs:

`uid`, `student`, `c29sdXRpb24=` and `submission`. The `uid` is a unique identifier composed of the date and the time of the submission. It is used to identify the ownership of the submitted code. The value of `student` is another JSON object containing the *student* files. The files' name and their corresponding contents serve serve as the JSON name/value pairs. Similarly, the values of `c29sdXRpb24=` and `submission` are JSON objects consisting of the *solution* and *submission* files. The `c29sdXRpb24=` is the `Base64` encoded string of the word *solution*. It serves as a flag to let CodeCheck know that the *solution* files' content is encoded. The encoding is necessary because the CloudCoder infrastructure sends the submission which is the JSON object from the client. Listing 5.1 shows an example JSON string for the "hello world" program in the C language. The JSON strings have new lines inserted to improve the visualization.

```
1  {
2    "uid":"2016.11.19.14.18.20.97",
3    "student": {
4      "hello.c": "#include <stdio.h>\nint main(void) {\n\t//
         ↪  TODO - add your code here\n\treturn 0;\n}"
5    },
6    "c29sdXRpb24=": {
7      "hello.c":"I2luY2x1ZGUgPHN0ZGlvLmg+CgppbnQgbWFpbih2
         ↪ b2lkKSB7CgkvLyBUT0RPIC0gYWRkIHlvdXIgY29k
         ↪ ZSBoZXJlCiAgICBwcmludGYoIkhlbGxvLCB3b3Js
         ↪ ZCIpOwoJcmV0dXJuIDA7Cn0="
8    },
9    "submission": {
```

```
10      "hello.c":"#include <stdio.h>\nint main(void) {\n\t//
          ↪ TODO - add your code here\n    printf(\"Hello,
          ↪ world\");\n\treturn 0;\n}"
11    }
12  }
```

Listing 5.1: JSON from CloudCoder to CodeCheck

The returned string from CodeCheck is shown in Listing 5.2. The JSON object has the `metaData`, `score` and `sections`. Only the `uid` inside the `metaData` value is used to identify the problem's submitter. Typically, a `score` of *0* indicates that there is a compilation error. Otherwise the `score` is in the form of a ratio. The `sections` contains the information about the exercise and its test results. There are 5 `types` of CloudCoder exercises and they are run, unitTest, tester, call and substitution. Different `type` of problem produces different content in the `sections`. The parsing methodology for each `type` of exercise is implemented in the `parseJSON()` method of the `OOPCodeCheckSubmission` class.


```
1   {
2     "metaData": {
3       "Elapsed": "1205 ms",
4       "ID": "hello",
5       "Level": "1",
6       "Problem": "16111922261842942241786536456",
7       "Submission": "submission",
8       "Time": "2016-11-19T22:26:36Z",
9       "uid": "2016.11.19.14.18.20.97"
```

```
  },
  "score": "1/1",
  "sections": [
    {
      "runs": [
        {
          "args": [
            {
              "name": "Command line arguments",
              "value": ""
            }
          ],
          "html": "<p><b>Input:</b></p><pre></pre><p><b>Output:</b></p><pre>Hello, world</pre>",
          "input": "",
          "output": "Hello, world",
          "passed": true
        }
      ],
      "type": "run"
    }
  ]
}
```

Listing 5.2: JSON from CodeCheck to CloudCoder

CHAPTER 6

Results

6.1 Enhanced CodeCheck Deployment

The enhanced CloudCoder was deployed on an Elastic Compute Cloud (EC2) instance hosted by Amazon Web Services (AWS). The instance has one hyperthreaded Intel Xeon E5-2676 v3 processor and 8 gigabytes (GB) of random access memory (RAM). Appendix A has the details on compiling and setting up CloudCoder.

Three classes at San Jose State University participated in the study during the 2016 Fall term. Programming in Java (CS49J) and Programming Paradigms (CS152) are from the Computer Science undergraduate program, and the Data Structures and Algorithms in C++ (CMPE180) is a graduate Computer Engineering course. The Programming in Java class had two sections with 30 and 25 students respectively. The Programming Paradigms class also had two sections with 34 and 29 students respectively. Only section 1 from both CS49J and CS152 had access to the enhanced CodeCheck exercises. Section 2 for both courses served as the control group. There was only one section of Data Structures and Algorithm in C++ class and it had 121 students. Enhanced CodeCheck exercises in Java and in C++, respectively, were given in the CS49J and CMPE180 classes. Exercises in Racket and JavaScript were assigned in the CS152 class.

6.1.1 CS49J - Programming in Java

There were a total of 53 exercises given during a span of nine weeks. Using the data from the 38 exercises given in the last seven weeks only, the average number of attempts per exercise and the average number of minutes between each attempt

are shown in Figures 37 and 38. As the topics of the exercises increased in complexity, a downward trend in the number of attempts and in the number of minutes between attempts indicate that the students are getting more comfortable with the Java language.

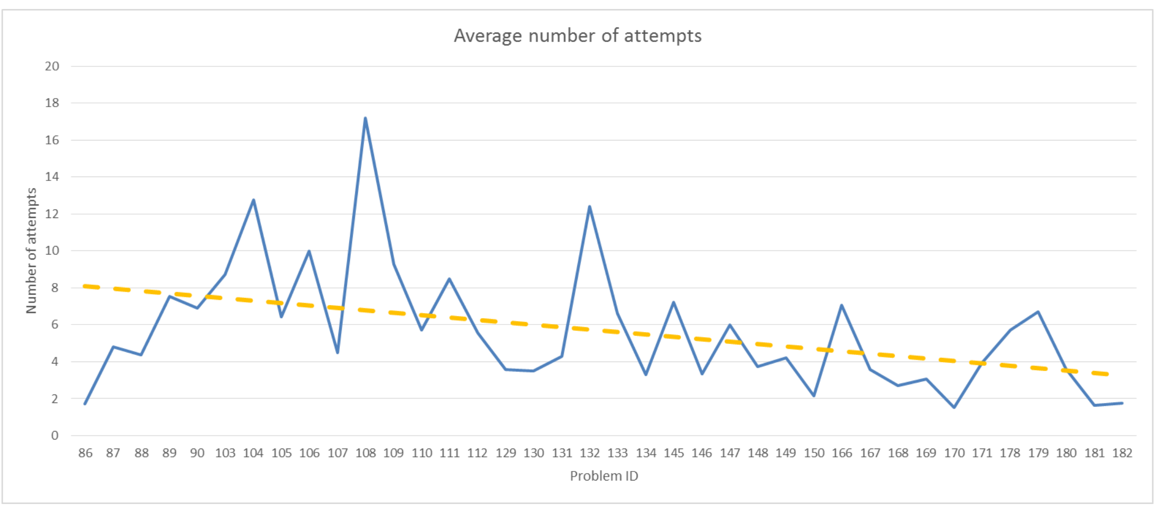


Figure 37: CS49J - Average number of attempts

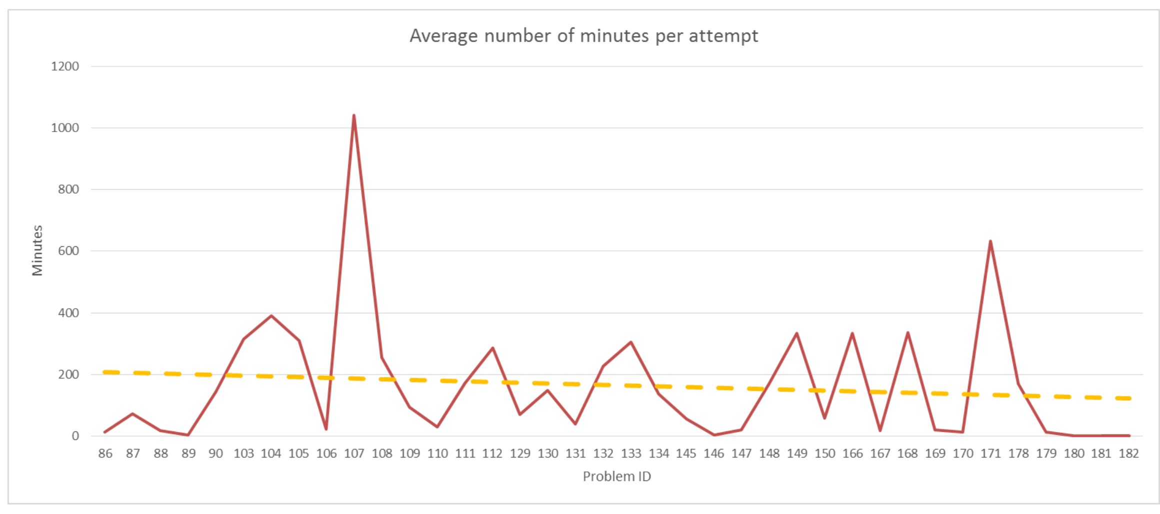


Figure 38: CS49J - Average number of minutes per attempt

In Figure 38, the peaks at problem numbers 107 and 171 correspond to relatively advanced and challenging exercises. Problem number 107 requires the student to

write a factory method. Problem number 171 challenges the student to write a list operation method without using loops and streams.

Two projects were given to students in both sections. The average scores are shown in Figure 39 below. The improvement between the projects as measured by the delta in the scores for section 1 is higher than section 2. This supports the conclusion that the students who were exposed to enhanced CodeCheck exercises are getting more comfortable with the Java language.

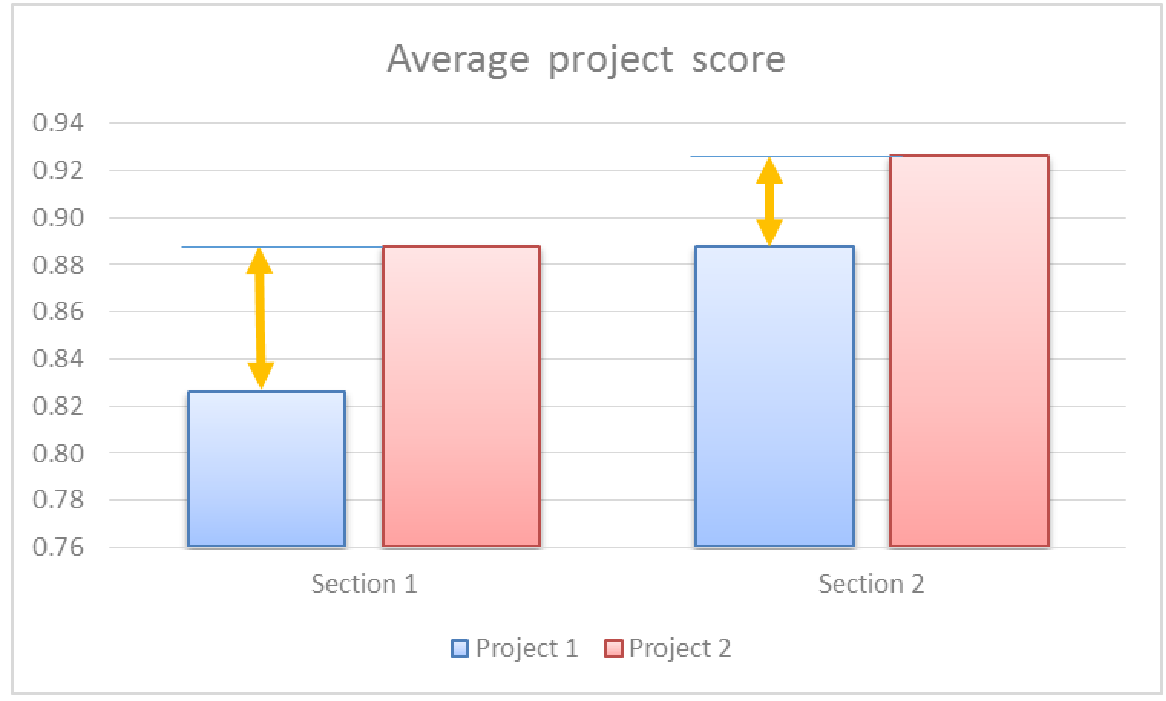


Figure 39: CS49J - Average project scores

6.1.2 CS152 - Programming Paradigms

There were a total of 41 exercises given during a span of six weeks. Using the data from the 30 exercises given in the last five weeks only, the average number of attempts per exercise and the average number of minutes between each attempt are shown in Figures 40 and 41. The slight increase in the number of attempts was caused

by the switch between Racket to the JavaScript language. However, the downward trend of the average number of minutes between attempts indicates that the students were not having difficulty in solving those problem.

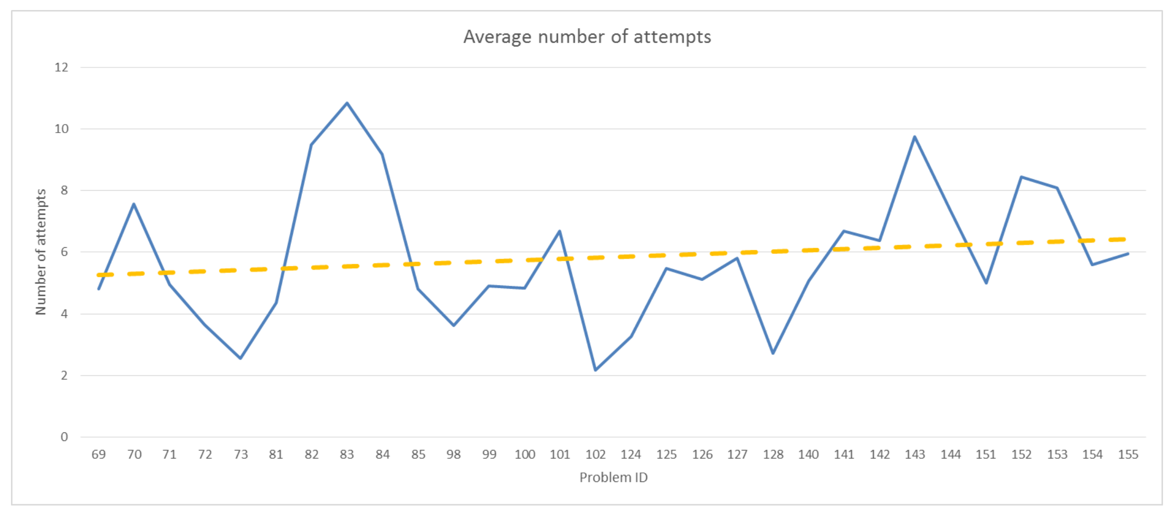


Figure 40: CS152 - Average number of attempts

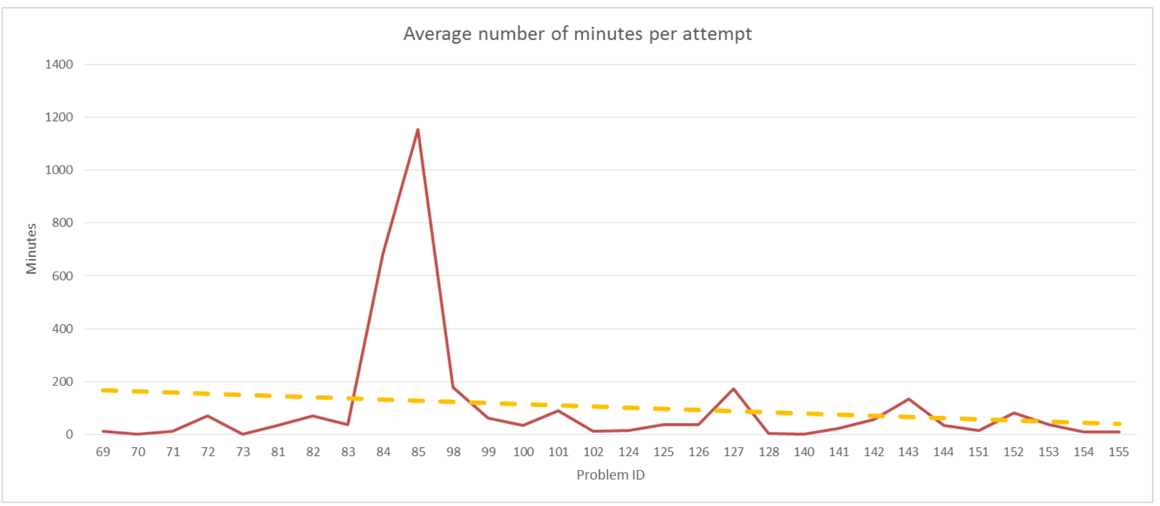


Figure 41: CS152 - Average number of minutes per attempt

In Figure 41, the spike corresponds to a Racket `foldl` and `foldr` exercise where the odd elements of a list are returned.

At the end of the study period, an exam was given to students in both sections.

The exam contained five Racket questions on recursion, tail recursion, higher-order functions, macros and contracts. The percentage of students that correctly answered these questions from both sections are shown in Figure 42. Overall, section 1 has a higher percentage of students with the correct answers. The two largest differences in the percentage are with the higher-order functions and the macros. Both of these two topics were covered extensively by the exercises. Also, the smallest difference in the percentage is with the contracts and this topic was not covered by the exercises.

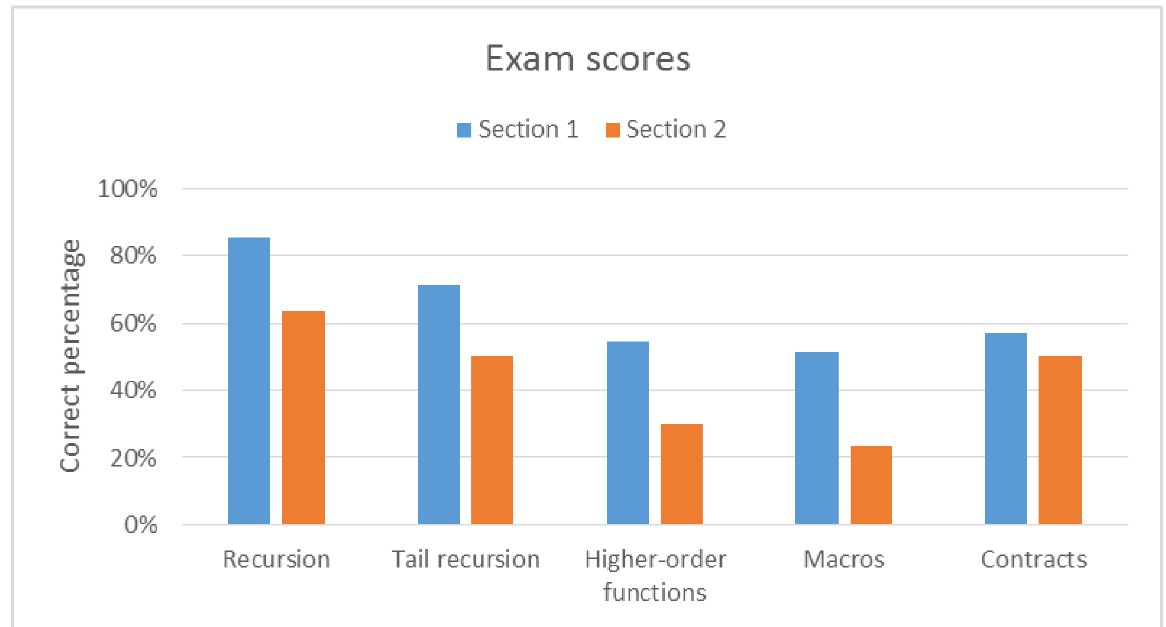


Figure 42: CS152 - Exam scores

To confirm the observation from the exam, an in-class quiz on Racket and JavaScript were given to both sections. The quiz had five questions on recursion and higher-order functions. One question is on Java and it was used as a control question. The grade distribution of both sections is shown in Figure 43. Out of 16 points, on average, the scores are 10.5 and 8.6 for section 1 and 2 respectively. If the control question scores are excluded, the averages drop to 8.0 and 6.1 respectively. Using the control question to filter out the students who could not answer it correctly,

the average for section 1 and 2 increases to 9.3 and 7.3 respectively. The filtered score distribution is shown in Figure 44. This result confirmed the previous observation on the exam that the students who had exposure to enhanced CodeCheck exercises did better in their class.

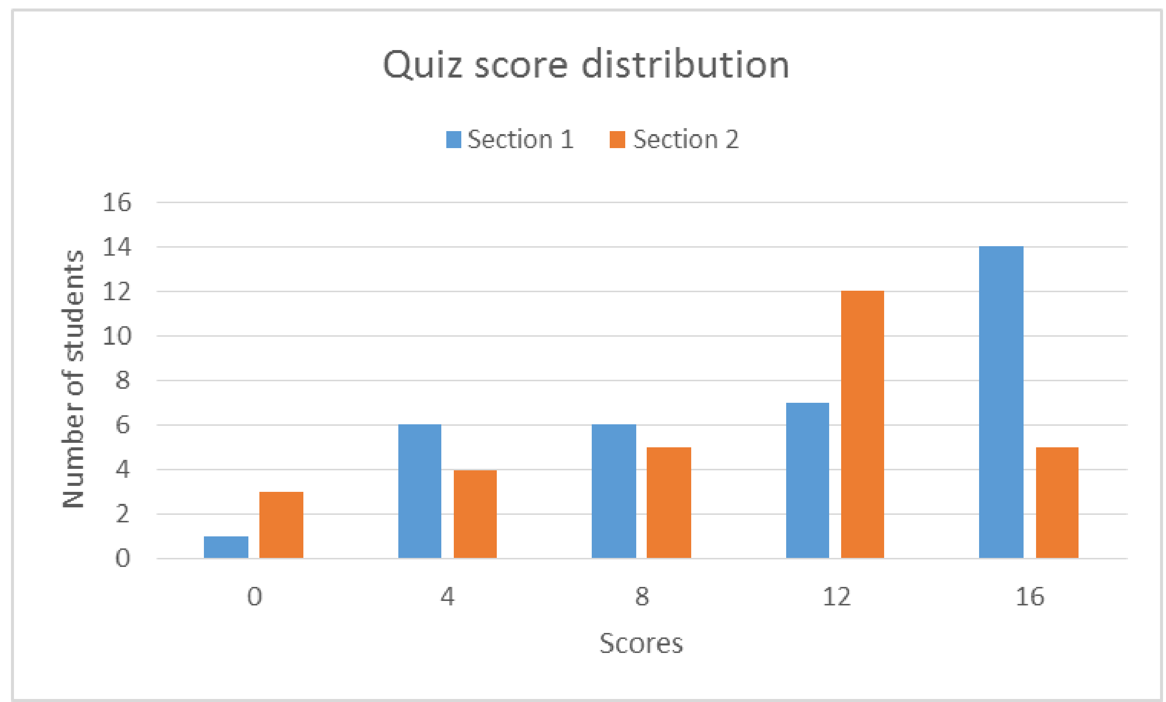


Figure 43: CS152 - Quiz scores

6.1.3 CMPE180 - Data Structures and Algorithms in C++

There were a total of 61 exercises given during a span of eleven weeks. Using the data from the 51 exercises given in the last nine weeks only, the average number of attempts per exercise and the average number of minutes between each attempt are shown in Figures 45 and 46. Both graphs show a downward trend with a tightening of the standard deviation.

In Figure 45, problems numbers 115 through 119 have the lowest number of attempts and these correspond to implementing methods of the `BankAccount` class

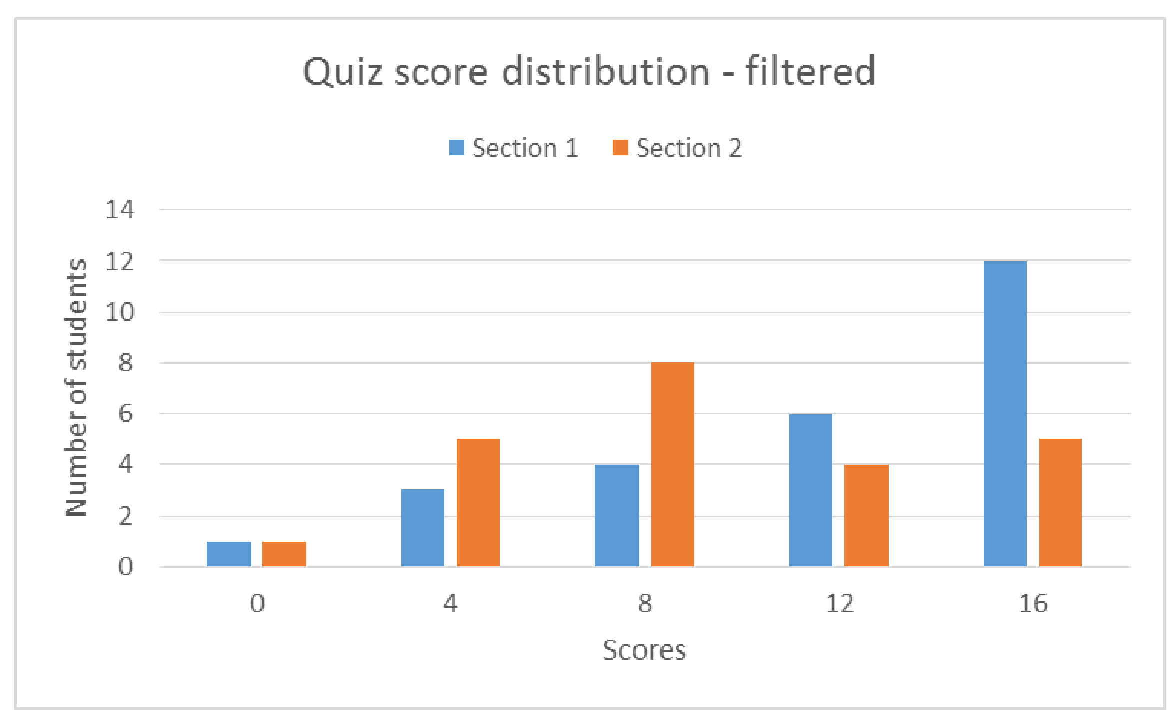


Figure 44: CS152 - Quiz scores filtered

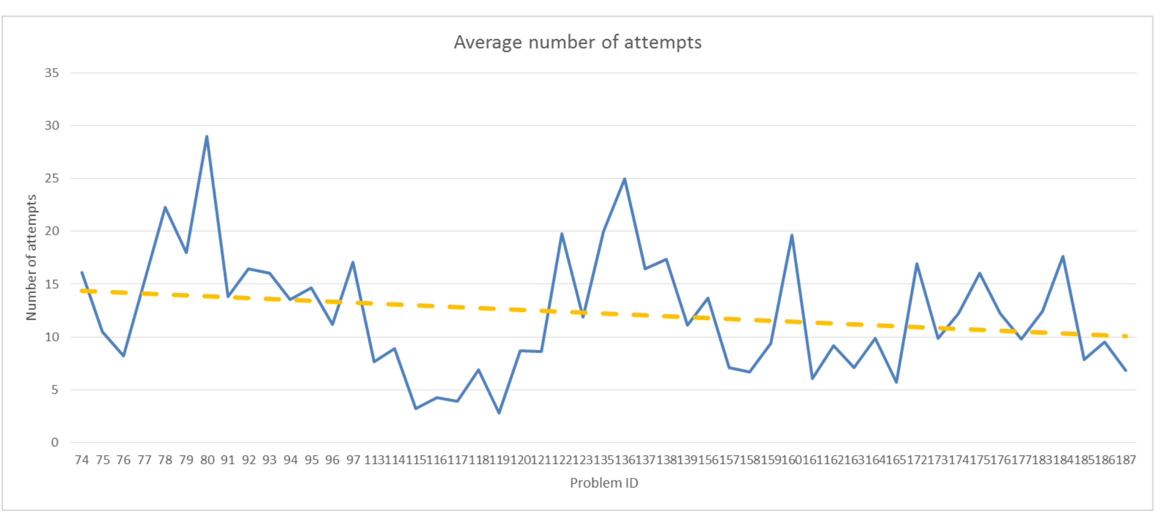


Figure 45: CMPE180 - Average number of attempts

and operators of the `Time` class.

Figures 47 and 48 show the grade distribution of the exam and quizzes against the CloudCoder exercise completion percentage. From these data, it seems to indicate

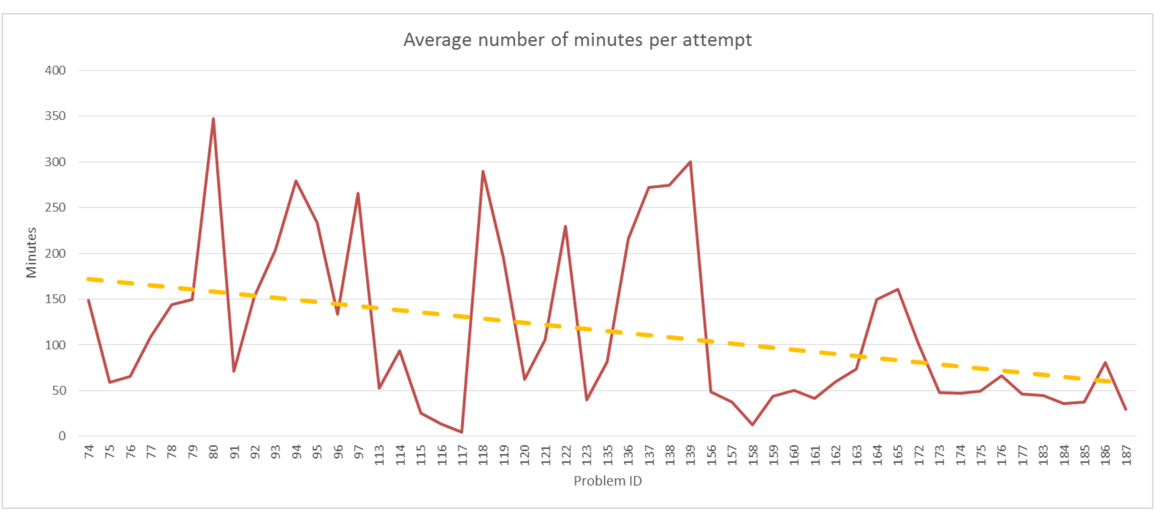


Figure 46: CMPE180 - Average number of minutes per attempt

that in order to get higher than 80% on the exam and quizzes, the student has to complete at least 60% of the exercises.

6.2 Enhanced CodeCheck Evaluation

The students from the three classes that participated in the study answered an online survey on the effectiveness, usability and applicability of the enhanced CodeCheck. Figures 49 through 51 have the distribution of the answers from 172 students.

As shown in Figure 49, more than 80% of the students agree that this tool is effective in helping them learn and master the language. If only the questions about homework and exam are considered, the agreement percentage is increased above 93%. Not all the classes have in-class assignments so the question about in-class assignment could be interpreted incorrectly.

Figure 50 shows that the students find the tool easy to use but only 58% of them used the GUI. Among the reasons of using a traditional IDE, the most common one

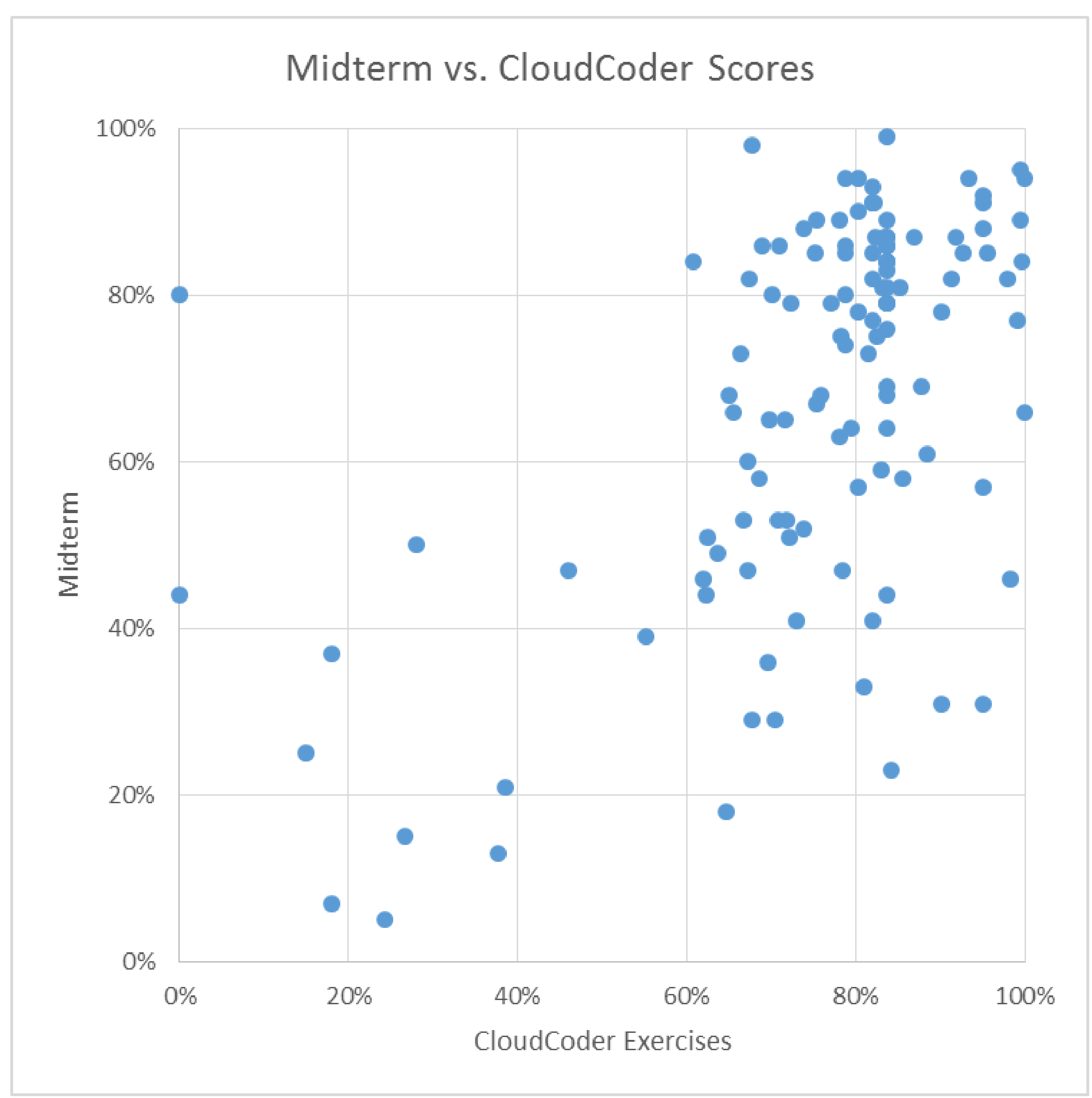


Figure 47: CMPE180 - Exam score

is the debuggability with enhanced CodeCheck. Also, there were other reasons such as the wait time for the submission results and the lack of a detailed failure output. More than half of the students find that the exercises given were difficult and 75% of them spent less than 6 hours to complete the weekly assignment.

Figure 51 shows that more than 85% of the respondents think that this tool could

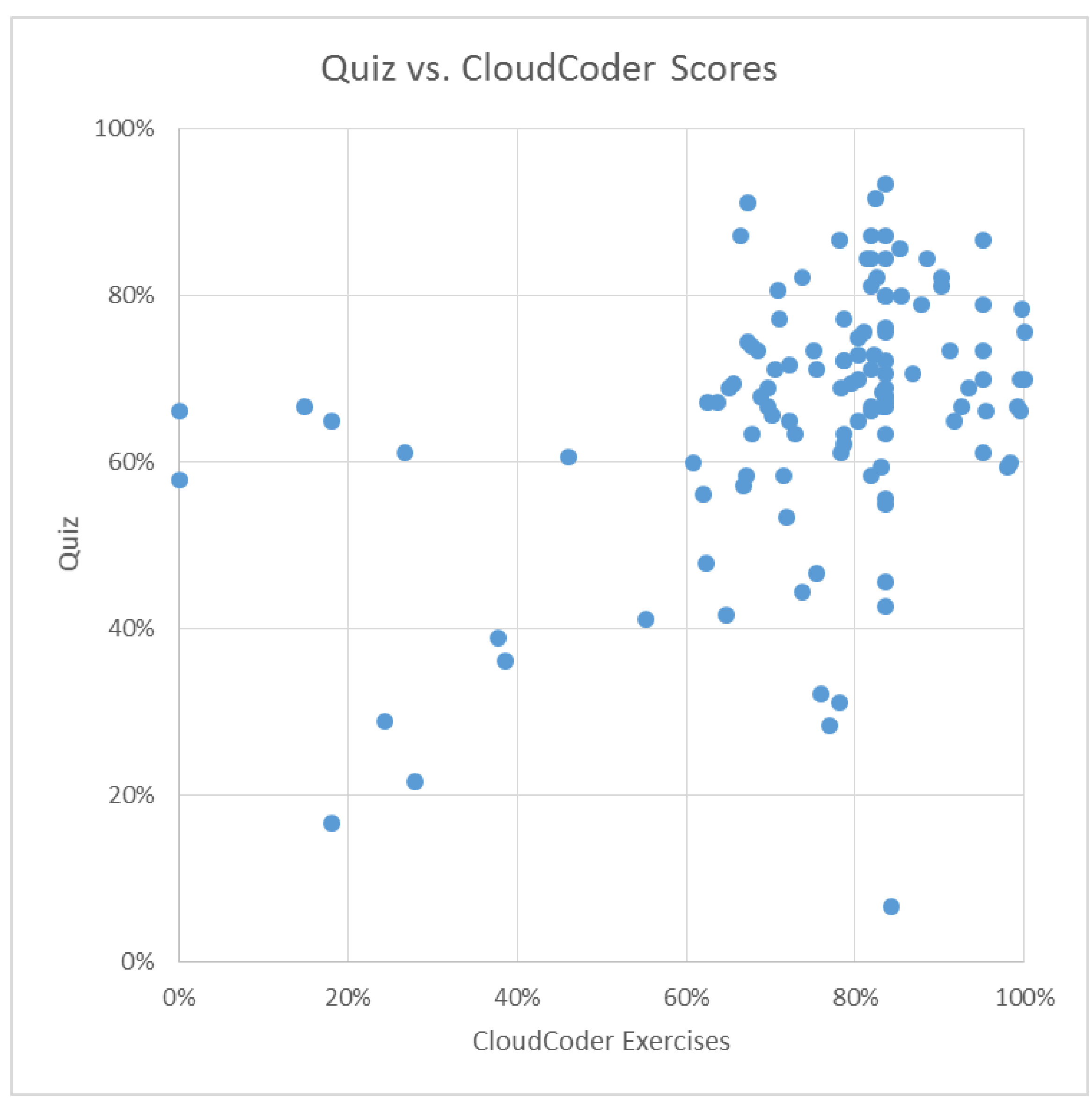


Figure 48: CMPE180 - Quiz scores distribution

be helpful in other programming classes. They would recommend this tool to other students. Most of the weekly exercises came with six or fewer questions and 96% of the students feel that this is sufficient for them.

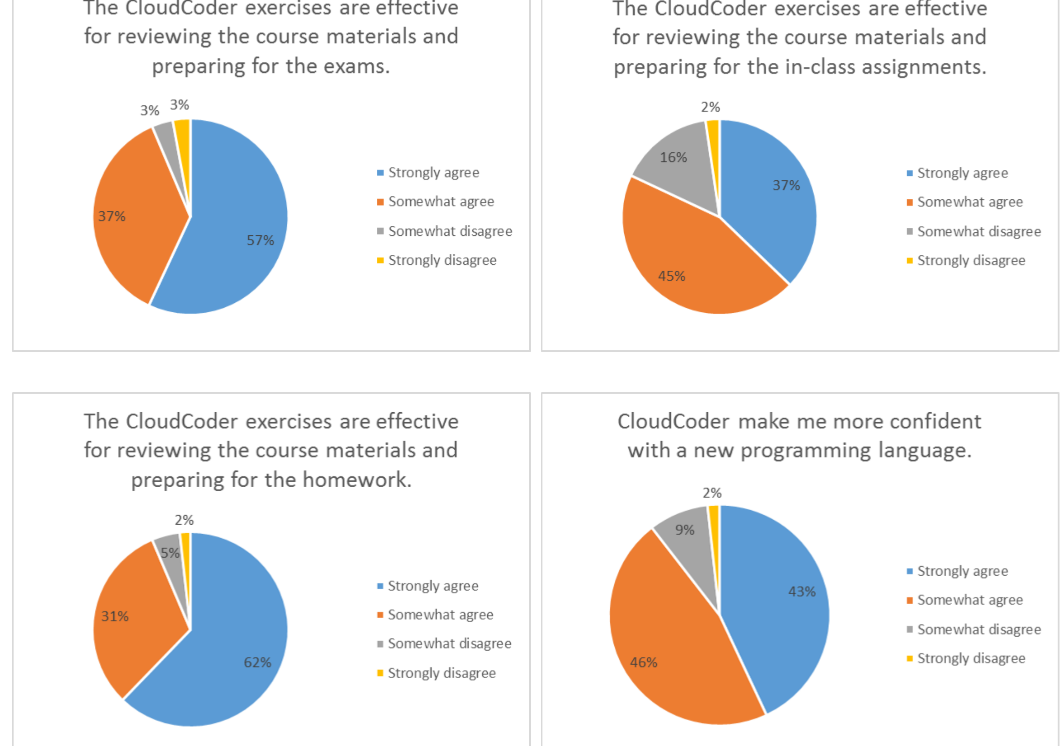


Figure 49: Questions on the effectiveness of the enhanced CodeCheck

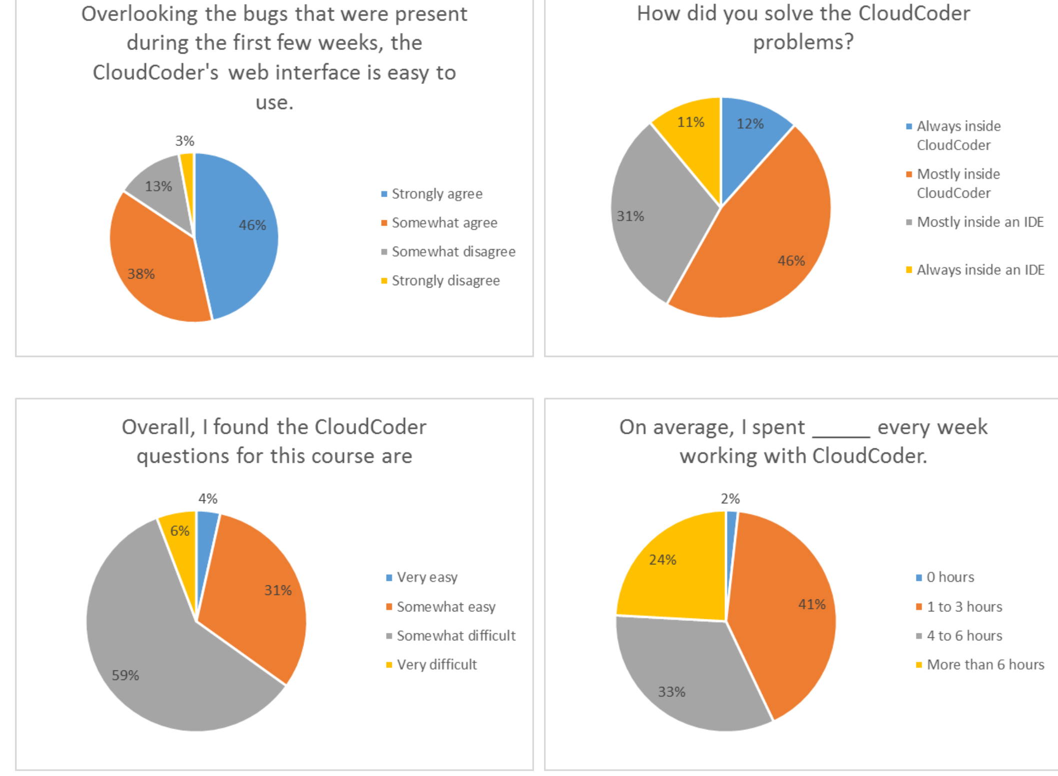
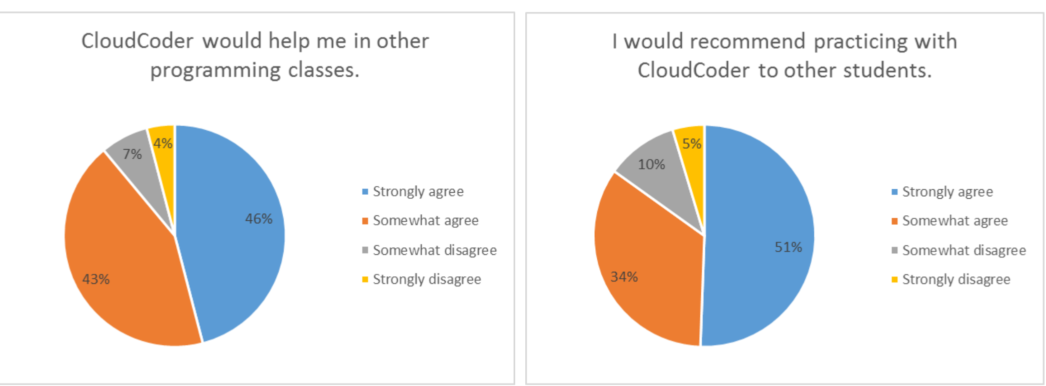


Figure 50: Questions on the usability of the enhanced CodeCheck



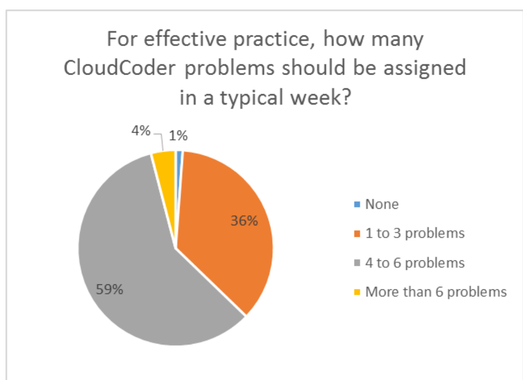


Figure 51: Questions on the applicability of the enhanced CodeCheck

CHAPTER 7

Deployment Challenges

There were several challenges encountered through the course of this project. All except the last one have been addressed. In chronological order, the challenges are described below.

Initially, the virtual machines were provided by the Information Technology (IT) department at San Jose State University. As part of the university resources, these machines were set up behind a firewall and there is a very strict security policy in opening Transmission Control Protocol (TCP) ports for external access. For development, a browser on the localhost would also enable access to these ports. However, the machines did not have a Graphical User Interface (GUI) like X Window System so internal access to the TCP ports were also not possible. These restrictions largely hindered CodeCheck and CloudCoder developments and deployments because users interact with them through web browsers, i.e, TCP ports.

As discussed in Appendix A, CloudCoder access is done through the Hypertext Transfer Protocol (HTTP) over Secure Sockets Layer (SSL), i.e., TCP port 443. However, the virtual machines were setup to only have TCP ports 80 and 8080 opened for external accessed. So in order to properly setup CloudCoder, its configuration was changed from port 443 to 80. With the reverse-proxy setup requirement for CloudCoder, the server inadvertently became an open web-proxy. This security hole was exploited and thousands of library articles were downloaded illegally. When this was discovered, the access to the machines was disable immediately. The incident also revealed a weakness that any Internet Protocol (IP) address inside the firewall

are automatically trusted by the library system. In an academic environment where virtual machines are made available to students, stricter access protocol should be implemented to, at minimum, excludes the virtual machines.

Once the virtual machines were setup on AWS, small EC2 instances (1 core with 2 GB of RAM) were used to reduce the cost. However, this is sufficient for CloudCoder only if the number of simultaneous access is limited to ten or fewer students. During the deployment for all three classes, the number of simultaneous users was in excess of 100 students. This increased the load and memory requirement caused the server to perform very poorly. In order to support around 170 students, a fairly large RAM capacity host is needed. The monthly cost to maintain a large RAM host was around $380.

Due to the various issues listed above, the virtual machines had to be reinitialized several times. With these, the CloudCoder development environment also had to be redone. The availability of the dependent packages was a roadblock. One such dependency is the `CloudCoderLogging/lib/owasp-java-html-sanitizer.jar` which is fetched from `http://owasp-java-html-sanitizer.googlecode.com`. Even at the time of this writing, the file is still not accessible. To remove this dependency on the dependencies' availability, a copy of the dependent packages were checked into the repository at `web-based-ide/CloudCoder_deps`.

There were also several challenges in using GWT's RPC. They are all related to the asynchronous nature of the calls. When enhancing the CloudCoder's user-facing pages "Edit problem" (5.2.1) and "Development" (5.2.2) pages, there was a need to get *student* and *solution* data from the server. In the "Edit page" case, the page was designed to retrieve the `Problem` separately from the `TestCase`. So in order to also get CodeCheck's files, a triple nested RPC is used. When initializing

the "Development" page, the number of *student* files needs to be retrieved from the server in order to allocate the editors, and, therefore, an asynchronous answer will not work. To get around this, a maximum number is assumed and used to initialize the editors. During the initialization of the editors' content, the number of editors is reduced. Fortunately, typical CodeCheck exercises have fewer than ten files.

CloudCoder and CodeCheck communicate using JSON strings via HTTP. The CodeCheck's JSON serializer has a feature which drops the element when the element is empty or zero. This was not known so the CloudCoder's JSON deserializer had to be updated multiple times. Also, with HTTP, the servers are susceptible to cyber hacks. The Domain Name System (DNS) of the domain name provider was hacked causing the *play.codecheck.ws* and *cloudcoder.codecheck.ws* domain names for the CodeCheck and CloudCoder servers to become unavailable for multiple days. These domain names were used to ease the reinitialization of the virtual machines. With each reinitialization, the machine's IP address is different.

Students' logical errors such as infinite loops can delay the response from CodeCheck to CloudCoder resulting in a misinterpretation of the failure. There is no clear way to tell that the connection was lost or the response is delayed.

CHAPTER 8

Conclusion

Ace and CodeMirror web-based code editors were successfully integrated with CodeCheck using JQuery functions. Both editors support IDE-like features such as syntax highlighting, bracket matching, auto-completion, auto-indentation and many others. Ace is better than CodeMirror in that it supports the Java language directly. The Ace editor is currently deployed at `http://horstmann.com/codecheck/index.html`.

The CloudCoder was also successfully integrated with CodeCheck. CloudCoder was enhanced to support multiple CodeCheck's *student* and *solution* files. The GUIs were also updated to support multiple Ace editors. The CloudCoder's builder was replaced with CodeCheck and JSON formatted strings were chosen as the communication protocol between CloudCoder's webapp and CodeCheck. The enhanced CodeCheck was tested and evaluated by 172 students from three different level of computer programming classes. With a total of 161 exercises and more than 7400 submissions, the enhanced CodeCheck is stabilized and can be used for a larger deployment.

Data collected during the evaluation show that students' performances improved with exposure to the enhanced CodeCheck exercises. In the CS152 class, the average scores on the exam and quizzes were found to be higher for the section where enhanced CodeCheck exercises were assigned. In CS49J, the improvement in projects' average score was found to be higher for the section with exposure to the enhanced CodeCheck exercises. From the survey, more than 80% of the students that participated in the

study believed that the exercises helped them in mastering the language. However, only 58% of them use the enhanced CodeCheck GUI when working on the exercises. And, more than 85% of the students believe that the enhanced CodeCheck is beneficial to have in other programming classes.

LIST OF REFERENCES

[1] Ace, https://ace.c9.io/#nav=about

[2] CodeMirror, https://codemirror.net/

[3] F.A. Deeb and T. Hickey, The Spinoza code tutor: faculty poster abstract, *Journal of Computing Sciences in Colleges*, 30(6):154–155, 2015

[4] S.H. Edwards, Work-in-Progress: Program Grading and Feedback Generation with Web-CAT, *Proceedings of the first ACM conference on Learning @ scale conference*, 215–216

[5] S.H. Edwards and M.A. Perez-Quinones, Web-CAT: Automatically Grading Programming Assignments , *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, 328–328

[6] Etherpad, http://etherpad.org/

[7] M. Goldman, G. Little and R.C. Miller, Collabode: collaborative coding in the browser, *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, 65–68

[8] C. Horstmann, CodeCheck, https://bitbucket.org/cayhorstmann/codecheck/

[9] C. Horstmann, Writing a Codecheck Problem, http://horstmann.com/codecheck/authoring.html

[10] D. Hovemeyer, CloudCoder, https://github.com/daveho/CloudCoder

[11] D. Hovemeyer, AlternateBootstrap, https://github.com/cloudcoderdotorg/CloudCoder/wiki/AlternateBootstrap

[12] M.J. Hull, D. Powell and E. Klein, Infandango: automated grading for student programming, *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 330–330

[13] ICEcoder, https://icecoder.net/

[14] ICEcoder Github, https://github.com/mattpass/ICEcoder

[15] jQuery plugin for Ace, https://cheef.github.io/jquery-ace/

[16] B.L. Kurtz, J.B. Fenwick, R. Tashakkori, A. Esmaili and S.R. Tate, Active Learning During Lecture Using Tablets, *Proceedings of the 45th ACM technical symposium on Computer science education*, 121–126

[17] B. Northrop, Parallel Asynchronous Calls in GWT, `http://www.summa.com/blog/2010/11/29/parallel-asynchronous-calls-in-gwt`

[18] D. Pritchard, Websheets: A Templated Online Coding Exercise System, *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 335–335

[19] W. Toll, Top 48 Integrated Developer Environments (IDEs) & Code Editors, `https://blog.profitbricks.com/top-integrated-developer-environments-ides/`

[20] Wikipedia, Comparison of integrated development environments, `https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments`

APPENDIX A

CloudCoder Compilation and Installation

The project's source codes can be download from Bitbucket via a secure shell (SSH) and `git@bitbucket.org:hienvu/web-based-ide.git`. The enhanced CodeCheck is in the `CloudCoder` subdirectory.

A.1 Prerequisites

The following packages are necessary to deploy CloudCoder [11]:

1. Java Development Kit (JDK) version 1.6 or later

2. MySQL relational database management system version 5 or later

3. Apache HTTP server version 2 or later

These packages are expected to be installed before compiling CloudCoder.

A.2 Compiling CloudCoder

The following instruction assumes that the Linux host has the necessary packages installed to clone source files from Bitbucket and to compile Java programs. CloudCoder is built using Google Web Toolkit (GWT) so a sofware development kit (SDK) version 2.6 or later is needed. With the assumption that the current working director is `web-based-ide/CloudCoder`, here are the steps to build CloudCoder:

1. Get the dependencies listed in `default.deps` file. The `fetchdeps.pl` is provided to download all the dependencies to the `deps` subdirectory and then copy

the dependencies to their expected locations. A copy of all the dependencies are also available inside the repository's `web-based-ide/CloudCoder_deps` directory. So it is also possible to copy them from that directory to the `deps` subdirectory and run the `fetchdeps.pl`. The script will by-pass the download portion and simply copy the dependencies to their expected locations.

2. Configure the CloudCoder by running the `configure.pl` script. Information about the GWT SDK, the MySQL and the Apache HTTP servers are needed to complete this step. The configuration parameters can be saved into a file with the `cloudcoder.properties` as the default name. If the file is present when the `configure.pl` script is executed, it will detect it and those parameters can be setup as default values.

3. Compile CloudCoder withe the `build.pl` script.

A.3 Deploying CloudCoder

A database needs to be created for CloudCoder inside the MySQL server. This can be done using the `org/cloudcoder/app/server/persist/CreateWebappDatabase` application. The following jar files need to be added to the classpath of the Java Virtual Machine (JVM):

- CloudCoderModelClassesPersistence/cloudcoderModelClassesPersist.jar
- CloudCoderModelClasses/cloudcoderModelClasses.jar
- deps/*

The Apache HTTP server needs to be setup with Secure Socket Layer (SSL)

enabled. With SSL, the server can be setup as a reverse-proxy between the users and the webapp. On Ubuntu Linux, these can be setup following these steps

1. Enable SSL with `sudo a2enmod ssl` and `sudo a2ensite default-ssl`

2. Add the following lines inside the `<VirtualHost _default_:443>` block of the `/etc/apache2/sites-enabled/default-ssl.conf` file

```
ServerName 198.199.106.137:443
ProxyPass /cloudcoder http://localhost:8081/cloudcoder
ProxyPassReverse /cloudcoder http://localhost:8081/cloudcoder
<Proxy http://localhost:8081/cloudcoder>
Order Allow,Deny
Allow from all
</Proxy>
```

   The 198.199.106.137 number is the IP address of the host.

The CloudCoder's webapp can be started by running this jar file `CloudCoderWebServer/cloudcoderApp.jar` with the `start` option. The `shutdown` option is used to stop the server.

To setup a new course, the webapp need to be run with the `createcourse` option. The transcript below shows how the Object-Oriented Design class was created for Fall 2016 term.

```
$ java -jar webapp/cloudcoderApp.jar createcourse
Create a CloudCoder course
What term?
0 - Winter
1 - Spring
2 - Summer
```

```
3 - Summer 1
4 - Summer 2
5 - Fall
[Enter value in range 0..5] 5
What year? 2016
Course name (e.g., "CS 101")? "CS 151"
Course title (e.g., "Introduction to Computer Science")? "Object-
    ↪ Oriented Design"
Course URL?
Instructor username? cay
What section is this instructor teaching (integer)? 5
Add another instructor? (y/n)y
Instructor username? hien
What section is this instructor teaching (integer)? 5
Add another instructor? (y/n)n
Success!
```

Students can be registered for a course via the user management page as shown in Figure 18. Students can be registered individually or all together at the same time via the 'Bulk register' button. The group registration can also be done at the command line with the webapp's `registerstudents` option. A tab-separated file containing the list of students is needed for the group registration. Each line in this text file contains the following information of each student: user name, first name, last name, email address, password and section number. The transcript below shows how students were registered for the Object-Oriented Design class.

```
$ java -jar webapp/cloudcoderApp.jar registerstudents
For which course would you like to register students?
0 - CCDemo - CloudCoder demo course
1 - "CS 49J" - "Programming in Java"
```

```
5  2 - "CodeCheck" - "CodeCheck problem type"
6  3 - "CS 151" - "Object-Oriented Design"
7  [Enter value in range 0..3] 3
8  Enter the name of the file containing a tab-separated list student
     ↪ registration entries in this format:
9  username        firstname       lastname        email   password
     ↪         section
10 Usernames in the datbase will be re-used, but the names/email/
     ↪ password will not be updated,and users will not be registered
     ↪ for a course if they are already registered
11 [default: ] ==> /home/hvu/cs151_5.txt
12 Note that this may be a slow operation
13 Some logging results will be appended to logs/cloudcoder.log rather
     ↪ than echoed to stdout
14 Registered 40 students for "CS 151"
```

APPENDIX B

Source Code

```
1  <html>
2  <head>
3    <style type="text/css" media="screen">
4      #ExampleB {
5        height: 300px;
6        width: 600px;
7      }
8    </style>
9  </head>
10 <body>
11   <script src="ace/ace.js" type="text/javascript" charset="utf-8"></
       ↪ script>
12   <script src="ace/mode-java.js"></script>
13   <script src="ace/ext-language_tools.js"></script>
14 
15   <h1>Ace example</h1>
16   <p>Java code</p>
17   <div id="ExampleA">
18 import java.util.Scanner;
19 public class ExampleA
20 {
21   public static void main(String[] args)
22   {
23     Scanner in = new Scanner(System.in);
24     boolean done = false;
25     while (!done)
26     {
```

```
27        System.out.println("Enter a number, 0 to quit");
28        int n = in.nextInt();
29        if (n == 0)
30          done = true;
31        else
32          System.out.println("The square is " + n * n);
33      }
34    }
35  }
36    </div>
37    <p>Java code with Ace</p>
38    <div id="ExampleB">import java.util.Scanner;
39  public class ExampleB
40  {
41    public static void main(String[] args)
42    {
43      Scanner in = new Scanner(System.in);
44      boolean done = false;
45      while (!done)
46      {
47        System.out.println("Enter a number, 0 to quit");
48        int n = in.nextInt();
49        if (n == 0)
50          done = true;
51        else
52          System.out.println("The square is " + n * n);
53      }
54    }
55  }
56    </div>
57    <script>
```

```
    ace.require("ace/ext/language_tools");
    var editor = ace.edit("ExampleB");
    editor.setTheme("ace/theme/eclipse");
    editor.getSession().setMode("ace/mode/java");
    editor.setOptions({
      autoScrollEditorIntoView: true,
      displayIndentGuides: true,
      enableBasicAutocompletion: true,
      enableLiveAutocompletion: false,
      enableSnippets: true,
      maxLines: 16,
      showInvisibles: true,
      tabSize: 2,
      useWorker: true
    });
  </script>
</body>
</html>
```

Listing B.1: Ace example

```
<html>
<body style="font-family: sans;">
  <link rel="stylesheet" href="codemirror-5.3/lib/codemirror.css">
  <link rel="stylesheet" href="codemirror-5.3/theme/neo.css">
  <link rel="stylesheet" href="codemirror-5.3/addon/hint/show-hint.
      ↪ css">
  <script src="codemirror-5.3/lib/codemirror.js"></script>
  <script src="codemirror-5.3/mode/css/css.js"></script>
  <script src="codemirror-5.3/addon/hint/show-hint.js"></script>
  <script src="codemirror-5.3/addon/hint/anyword-hint.js"></script>
  <script src="codemirror-5.3/addon/edit/matchbrackets.js"></script>
```

```
11    <script src="codemirror-5.3/addon/edit/closebrackets.js"></script>
12
13    <h1>CodeMirror example</h1>
14    <p>Java code</p>
15    <textarea id="ExampleA" rows="17" cols="66">
16  import java.util.Scanner;
17  public class ExampleA
18  {
19    public static void main(String[] args)
20    {
21      Scanner in = new Scanner(System.in);
22      boolean done = false;
23      while (!done)
24      {
25        System.out.println("Enter a number, 0 to quit");
26        int n = in.nextInt();
27        if (n == 0)
28          done = true;
29        else
30          System.out.println("The square is " + n * n);
31      }
32    }
33  }
34    </textarea>
35    <p>Java code with CodeMirror</p>
36    <textarea id="ExampleB" rows="17" cols="66">
37  import java.util.Scanner;
38  public class ExampleB
39  {
40    public static void main(String[] args)
41    {
```

```
42      Scanner in = new Scanner(System.in);
43      boolean done = false;
44      while (!done)
45      {
46        System.out.println("Enter a number, 0 to quit");
47        int n = in.nextInt();
48        if (n == 0)
49          done = true;
50        else
51          System.out.println("The square is " + n * n);
52      }
53    }
54  }
55    </textarea>
56    <script>
57      var editor = CodeMirror.fromTextArea(
58        document.getElementById("ExampleB"),{
59          extraKeys: {"Ctrl-Space": "autocomplete"},
60          lineNumbers: true,
61          matchBrackets: true,
62          autoCloseBrackets: true,
63          theme: 'neo',
64          tabSize: 2
65       });
66    </script>
67  </body>
68  </html>
```

Listing B.2: CodeMirror example