

**HIGHER COLLEGE OF TECHNOLOGY**

**Department of Information Technology**

**Software Engineering Specialization**

**Programming Contest System**

In Partial Fulfillment of the Requirements for B.Tech Degree With Specialization in Software Engineering

**Prepared By**

Assim Al-Marhuby: 16S0940

**Under the guidance of**

Mr. Mohammed Mushtaq

Semester 2 / 2014



**HIGHER COLLEGE OF TECHNOLOGY**

**Department of Information Technology**

**BONAFIDE CERTIFICATION**

Certified that the project report “Programming Contest System” is the bonafide work of “Assim Al-Marhuby” who carried out the project work under my supervision for the partial fulfillment of the requirements for “B.Tech” with specialization in “Software Engineering”.

HEAD OF THE DEPARTMENT SUPERVISOR

DATE:

**2. Table of Contents**

|  |  |  |
| --- | --- | --- |
| **SL. No** | **Chapter Title** | **Page No.** |
| - | Acknowledgement | 5 |
| - | Abstract | 6 |
| 1 | Introduction | 7 |
| 1.1 | Introduction | 7 |
| 1.2 | Problem Statement | 7 |
| 1.3 | Project Objective | 7 |
| 1.4 | Project Description | 8 |
| 1.5 | Action Plan | 9 |
| 2 | System Requirements Specification | 10 |
| 2.1 | Introduction | 10 |
| 2.1.1 | Purpose | 10 |
| 2.1.2 | Definitions, Acronyms and Abbreviations | 10 |
| 2.2 | Overall description | 10 |
| 2.2.1 | System Perspective | 10 |
| 2.2.2 | System Features | 10 |
| 2.2.3 | Operating Environment | 11 |
| 2.2.3.1 | Hardware Requirement | 11 |
| 2.2.3.2 | Software Requirement | 11 |
| 2.2.4 | Design/Implementation constraints | 11 |
| 2.2.5 | Assumptions and Dependencies | 11 |
| 2.3 | Specific Requirements | 12 |
| 2.3.1 | Functional Requirements | 12 |
| 2.3.2 | Non-Functional Requirements | 12 |
| 2.3.2.1 | Performance | 12 |
| 2.3.2.2 | Reliability | 13 |
| 2.3.2.3 | Security | 13 |
| 2.3.2.4 | Usability | 13 |
| 2.4 | External Interface Requirements | 13 |
| 2.4.1 | User Interfaces | 13 |
| 3 | System Design | 15 |
| 3.1 | Introduction | 15 |
| 3.2 | System Design Overview | 15 |
| 3.3 | Application Design Detail | 16 |
| 3.3.1 | Data Flow Diagrams | 16 |
| 3.3.2 | System Flow Diagram | 18 |
| 3.3.3 | Entity – Relationships | 19 |
| 4 | Implementation | 19 |
| 4.1 | Implementation | 19 |
| 4.1.1 | System Implementation Description | 19 |
| 4.1.2 | Coding | 20 |
| 4.1.3 | Validation Checks | 33 |
| 4.1.4 | Reports | 38 |
| 4.2 | Testing | 40 |
| 5 | Conclusion and Future | 42 |
| 5.1 | Summary of Findings | 42 |
| 5.2 | Conclusion | 42 |
| 5.3 | Future Scope | 44 |
| - | References | 45 |

**3. List of Figures**

|  |  |  |
| --- | --- | --- |
| **SL. No** | **Name of the diagram** | **Page No.** |
| 2.1 | Contestant Login Screen | 14 |
| 2.2 | Contestant Dashboard Screen | 14 |
| 3.1 | DFD Level 0 | 16 |
| 3.2 | DFD Level 1 | 17 |
| 3.3 | System Flow Diagram | 18 |
| 3.4 | ER Diagram | 19 |
| 4.1 | Validation – Configuration Screen IP Address | 34 |
| 4.2 | Validation – Contestant Login | 34 |
| 4.3 | Validation – Server can’t start | 35 |
| 4.4 | Validation – No connection to server | 36 |
| 4.5 | Validation – “Submit Code” button disabling | 37 |
| 4.6 | Validation – Contest time | 38 |
| 4.7 | Generate Report from Menu | 39 |
| 4.8 | PDF Report Example | 39 |

**4. List of Tables**

|  |  |  |
| --- | --- | --- |
| **SL. No** | **Name of the table** | **Page No.** |
| 4.1 | Black Box Testing | 41 |

**Acknowledgment**

I would like to express my special thanks of gratitude to my teacher Mr. Mushtaq as well as the Software Engineering section of the Higher College of Technology who gave me the golden opportunity to do this wonderful project titled as “Programming Contest System”.

I’m highly indebted to Mr. Mushtaq for his guidance and constant supervision as well as for providing necessary information regarding Course Project and also for his support in completing the project.

**Abstract**

Programming Contest System is a system that allows hosting programming contests and automates the judging process. The contestants and the contest admin have to be connected on the same Local Area Network (LAN), the contestants can send solutions of the problems which will be automatically compiled and validated by the Contest Admin PC.

The project was written entirely in Java and was designed in a complete object-oriented manner, it uses Java’s Remote Method Invocation (RMI) to transfer data in the network, it uses serialization to save objects’ data as files and load them back when necessary, and the Java Database Connectivity (JDBC) with the SQLite driver for storing and retrieving data, and most importantly it makes use of Threads to make the software multi-threaded and allow it to perform many tasks simultaneously.

The Programming Contest System can help host programming contests easily and can help encourage people to do programming as a hobby or profession since it improves the brain functions and makes people become better thinkers. So hopefully there will be more programming contests in Oman and other countries using this software since it automates a lot of the tasks and makes hosting one a very easy thing.

**Chapter 1**

**Introduction**

* 1. **Introduction:**

Programming Contest System is a system that allows hosting programming contests. When running the application for the first time, it’ll allow you to select whether to configure the PC as a Contestant or Contest Admin. In every contest, there can be only one Contest Admin, and many Contestants. When the contestants send their submissions, the Contest Admin PC will automatically receive the submission and compile and validate it and update the scoreboard based on the result. The Contest Admin will be in charge of the contest settings such as adding contestants, programming languages, and problems, and the contest time.

* 1. **Problem Statement:**

The problem is that there isn't an adequate number of programming competitions in Oman and sometimes it’s because there isn’t a platform to automate the process for that. There’s a lot of skilled programmers and problem solvers and they would wish if there was a place where they can compete to be better in what they do, but a problem is that it’s very rare in Oman, and there has to emphasis on the importance of programming contests because they help improve skills and understanding in a lot of things in life as well as it exercises all the hemispheres of the brains which will help people make better use of their brains and be better thinkers.

* 1. **Project Objective:**

The idea of this project is to create a platform for managing programming contests and it will help run them along with judging by compiling and running the submissions and generating the scoreboard automatically based on the submission results. The plan is also to publish the project as an open source project after the end of the semester when Course Project is completed, and this will not only help people use the application for free, but it will also help learners to learn more about programming and how the program was created, and it will also help them learn some important Java concepts used.

* 1. **Project Description:**

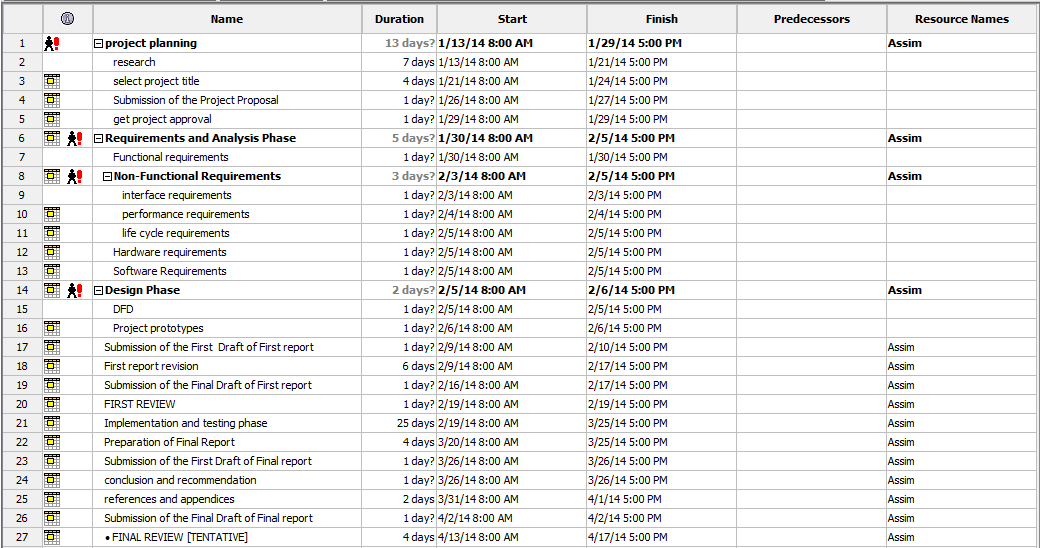
Computers will be connected to a Local Area Network (LAN) where each computer is considered to be a Contestant (Client), and one computer which will have the server interface which will be considered as the Contest Admin (Server). The Contest Admin will be in charge of creating logins for contestants and other settings for the contest. The Contest Admin will also be able to select allowed languages and compilation settings, as well as adding the problems and their correct solutions in the system.

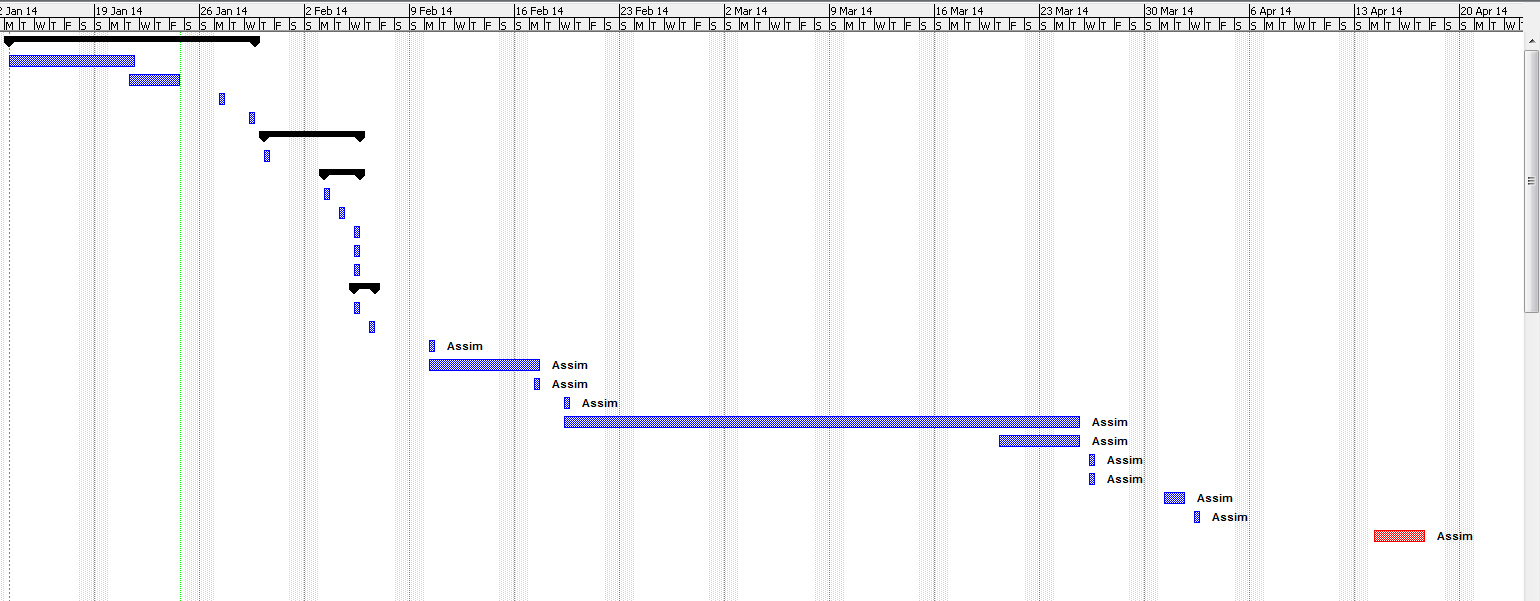
The contestants are going to be the clients and will login based on the credentials from the server and will connect to the server. Once the contest starts, they will be able to submit solutions to the problems and the Contest Admin PC will automatically compile, run and validate the answers and recalculate the scoreboard standings.

At the end of the contest, the standings will show based on who solved the most number of problems, if there is a tie, then the team that has less penalty is the team which will be in the lead, and penalty is nothing but the total number of failed submissions.

After all that, the Contest Admin can generate a report for the contest and its details such as standings and other statistics.

* 1. **Action plan:**

1. **



**Chapter 2**

**System Requirements Specifications**

**2.1 Introduction**

**2.1.1 Purpose**

The purpose of this document is to present a detailed description of the Programming Contest System. The requirements are documented in such a way that it breaks the deliverables into smaller components. The information is organized in such a way that we will not only understand the boundaries of the work that needs to be done, but also what functionality needs to be developed and in what order. It will explain the features of the system, what the system will do, the constraints under which it must operate, and the different types of requirements ranging from functional, non-functional, operating, hardware, and more.

**2.1.2 Definitions, Acronyms and Abbreviations**

client. The contestant who will be using the client interface of the system

server. The contest admin who will be using the server interface of the system.

**2.2 Overall Description**

**2.2.1 System perspective**

The Programming Contest System is a system that will be used to for programming contests and the idea of it is to encourage such contests in the society. Such systems do exist and this is as a replacement to these system by providing an easier interface and the use of multithreading to speed up certain tasks.

**2.2.2 System Features**

* Code submission for problems (questions) and validation with submission results.
* Support for virtually all programming languages as long as it can be compiled by the command line.
* Complete contest management from the server.
* Automatic code validation.

**2.2.3 Operating environment**

This will be able to run on multiple operating systems such as Windows, Mac OSX, Linux, Solaris and more. As long as the Java Virtual Machine is installed in those systems, then it will be capable of running the software.

**2.2.3.1 Hardware Requirement**

* Router/Switch
* Ethernet cable (Optional if connected wirelessly)
* PCs with NIC or WNIC to connect to the LAN.

**2.2.3.2 Software Requirement**

* Java Development Kit 7 or above for compiling and debugging.
* Java Virtual Machine 7 or above for running the software.
* Eclipse 4.3 IDE for developing the software.

**2.2.4 Design/Implementation Constraints**

* Requires a Java Virtual Machine (JVM) to run.
* A Local Area Network (LAN) has to be formed.
* Contestant disconnection from the server will not be known because the TCP/IP protocol wasn’t designed to identify whether the connection is alive or not.

**2.2.5 Assumptions and Dependencies**

* Contestants (Client) will not be able to login unless a Contest Admin (Server) has added the contestant to the system.
* Contestants (Client) will only be able to submit problems during the contest time set in the Contest Admin (Server).
* Clients should have a Network Interface Card (NIC) or Wireless Network Access Card (WNIC) to be able to connect to the network.
* All contestants and the Contest Admin should be connected to the same Local Area Network (LAN).

**2.3 Specific Requirements**

**2.3.1 Functional Requirements**

* The client should be able to send a submission to the server.
* The server should allow users login.
* The client should be able to login.
* The server should be able to change contest settings.
* The server should be able to add users to the system.
* The server should generate contest reports.
* The server should be able to add programming language along with their settings for compiling and executing.
* The server should be able to add problems along with the input and output for testing with the user program.
* The server should be able to automatically validate the result and send the result back to the client.
* The server should be able to restore the data in case of a software failure or restart.
* The client should be able to get the scoreboard.
* The server should automatically update the scoreboard after every validated submission.

**2.3.2 Non-Functional Requirements**

**2.3.2.1 Performance:**

* The system must be fast when processing user requests.
* The server software should not lag when there are many requests from the users.

**2.3.2.2 Reliability:**

* The system should be able to restore data in case of a crash of software failure.
* The system should have a reliability of 99%.

**2.3.2.3 Security:**

* Server should prevent unauthorized access to the system
* User should not allow the clients to access the database directly, just the data that is extracted from it.

**2.3.2.4 Usability:**

* System should be well designed and user-friendly.
* System should have help features on the application to help both the novice and advanced users.

**2.4 External Interface Requirements**

**2.4.1 User Interfaces**

**2.4.1.1 Contestant Login Screen**

The first screen is the Contestant Login screen, this is where the contestant will enter their username and password and will access the system.

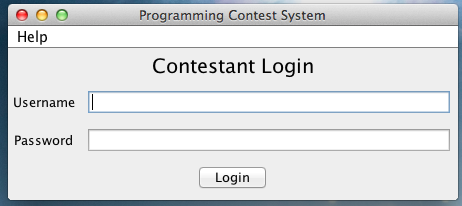


Figure 2.1. Contestant Login Screen.

**2.4.1.2 Contestant Dashboard Screen**

This is the screen the contestants will be seeing after they login, they will be able to know how much time is left in the competition. The contestants will also have a tab which will show all their submissions, and will allow them to submit more, and another tab for showing the scoreboard which will show the current standings of the contest.

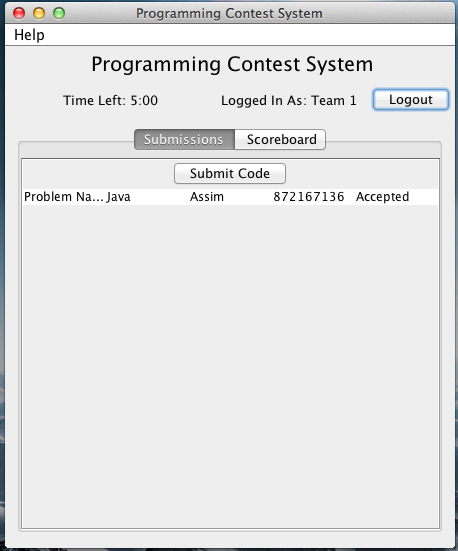


Figure 2.2. Contestant Dashboard Screen.

**Chapter 3**

**System Design**

**3.1 Introduction**

The purpose of this document is to describe the implementation of the Programming Contest System which is described in the Software Requirements Specifications. The Programming Contest System will help in conducting programming contests and making the judging procedure fast and automated. The scope of this software is to allow contestants to submit their solutions and the Contest Admin PC will automatically determine whether the solution is correct or not and based on the contestants’ generated output from their submitted programs.

**3.2 System Design Overview**

Database Interface: This module will create and connect to a SQLite database and modify it accordingly based on actions certain actions.

Code Runner Module: This module is in charge of compiling and running the submissions and it will work with the Code Validator Module to check the output.

Code Validator: This module will check the contestants’ submitted program output with the expected output (correct solution), and based on that it will decide on the contestant score and penalty.

Report Generator: This module will generate reports that are useful after the contest.

Scoreboard: The scoreboard module will check the contestants’ score, penalty, and submissions and based on that it will keep track of the contestants’ standings.

Contest Admin (Server): The Contest Admin is in charge of controlling the whole contest like the contest start time and other settings, it will receive submissions from contestants, and it will automatically calculate the results.

Contestant (Client): The contestant logs in and connects to the Contest Admin PC (Server), and will be able to submit and the server will do the processing and data storage.

**3.3 Application Design Detail**

**3.3.1 Data Flow Diagrams**

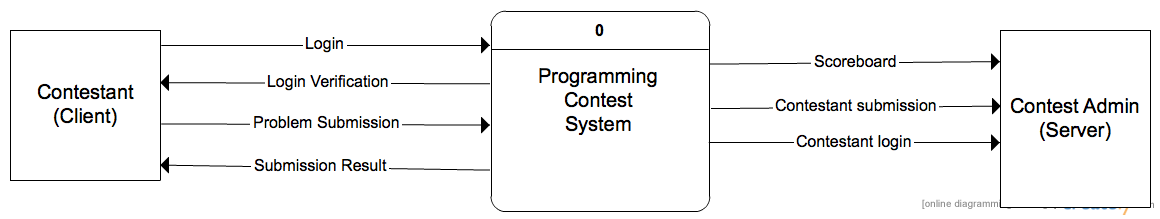


Figure 3.1. DFD Level 0

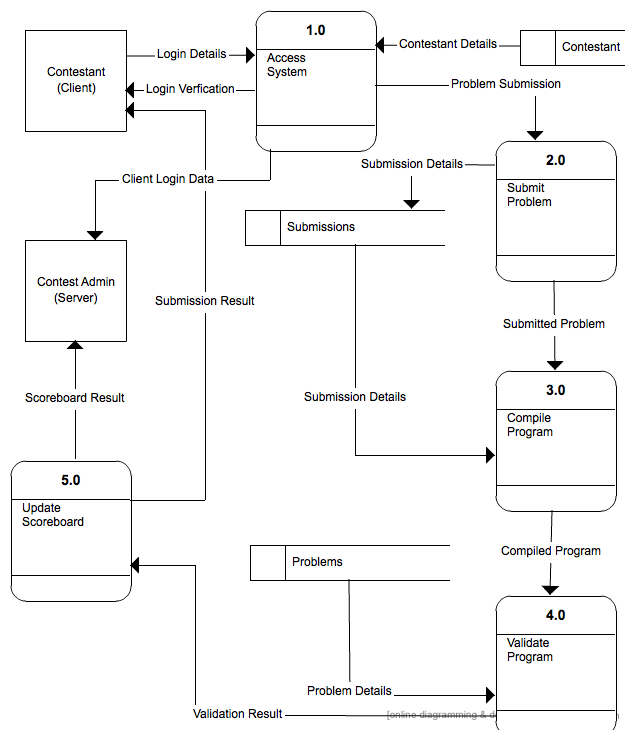


Figure 3.2. DFD Level 1

**3.3.2 System Flow Diagram**

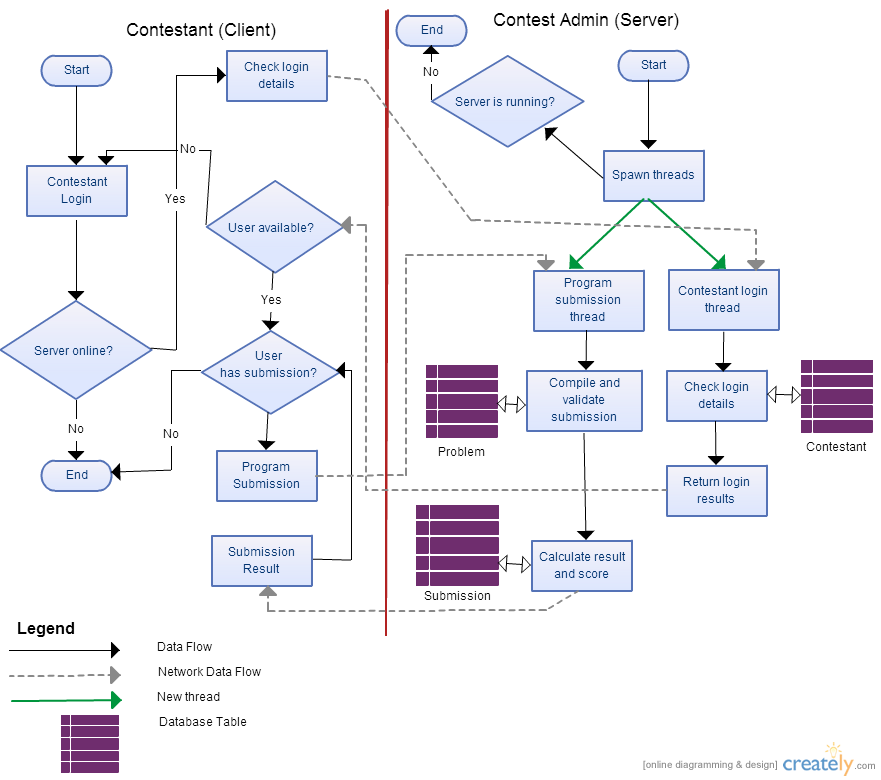
****

Figure 3.3 System Flow Diagram

**3.3.3 Entity - Relationships**

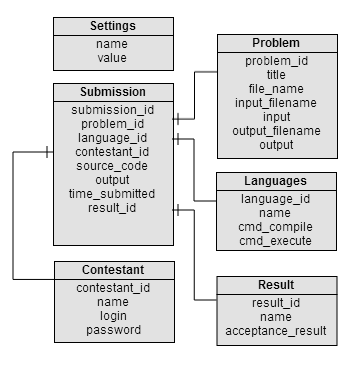


Figure 3.4. ER Diagram

**Chapter 4**

**Implementation**

**4.1 Implementation**

**4.1.1 System Implementation Description**

First of all, the project requirements and design was converted to sketches that explains how the system would look like and at the same time some flow lines were shown from one screen to another to give a mental image of how the actual system would look like and work.

The first major step was the object-oriented design. It was to convert every element in the project to a model class by defining its instance variables and methods. For example: Contestant class would have a username instance variable along with other variables that defines it. The process was then to combine similar ones together and remove unwanted ones that had no relevance in the project.

The second major step was creating the Database Adapter for the SQLite database. This class handled all the communications to the database. The Adapter had methods for performing Create, Read, Update, and Delete operations (CRUD) for every table. The data used as input or output for CRUD operations were given and taken from the Database Adapter was as an object from one of the model classes in the previous step. This made things more logical and easier in the whole development process.

Step number three was to design the views and screens for everything and by inserting dummy data to prototype it.

Later on, the actual coding and functionality was made, this is where the threads, connections, functionalities, and everything else was written and put together to complete the system.

**4.1.2 Coding**

In this section, some of the most important snippets from the 6000 lines of codes written will be outlined here, and the codes are fully commented so it will be easy to understand what each code does.

**4.1.2.1 Contestant Login Code**

The below code demonstrates when the Contestant tries to connect to the Contest Admin using Java Remote Method Invocation (RMI), and an object from the RmiData class holds all the data that is being transferred from the network between client and server and the other way around. The ConfigManager class is the class in charge of setting and getting configurations and automatically storing it in a file whenever there are changes to it. The detail of how the configuration file is saved and loaded won’t be clear in this snippet but it will be clear 4.1.2.5 section snippet where the details are explained.

/\*\*

\* Run when client tries to login. If info correct, show client and start

\* appropriate threads.

\*/

**public** **void** login(String username, String password) {

// Get instance of ConfigManager from file

ConfigManager cm = **new** ConfigManager();

**try** {

// Try to connect to RMI through port 1099

Registry registry = LocateRegistry.*getRegistry*(cm.getServerIp(),

1099);

Rmi rmi = (Rmi) registry.lookup("pcs");

// Send login data

RmiData socketData = rmi.getData(username, password);

// If login details are correct

**if** (socketData.getLoginResult()) {

// Store username and password for future updates

**this**.username = username;

**this**.password = password;

socketData.saveToFile(); // save Rmi data to file

// Show client view frame and close the login view

clientView = **new** ClientView(**this**, username);

clientView.getFrame().setVisible(**true**);

loginView.getFrame().dispose();

// Start updater thread

clientUpdater = **new** ClientUpdater();

clientUpdater.start();

} **else** {

// Login failed

JOptionPane.*showMessageDialog*(**null**,

"Login failed. Make sure of username and password.");

}

} **catch** (RemoteException | NotBoundException e) {

// Problem with the server connection

JOptionPane.*showMessageDialog*(**null**, "Could not connect to server.");

e.printStackTrace();

}

}

**4.1.2.2 Server Initialization Code**

This code demonstrates how the server starts the RMI service. The server also spawns two threads, one is the ServerUpdater which looks for new data periodically and updates the views if there are any changes in the data, and the other is the CodeRunner thread which looks for pending submissions and works on validating them.

/\*\*

\* The constructor for the server controller. It starts an RMI service and

\* spawns thread that updates the server and looks for code to compile and

\* execute.

\*/

**public** Server() {

// Show server view

serverView = **new** ServerView(**this**);

serverView.getFrame().setVisible(**true**);

// Start server and listen from port 1099

Rmi rmiService = **null**;

**try** {

rmiService = **new** RmiImpl();

Registry registry = LocateRegistry.*createRegistry*(1099);

registry.rebind("pcs", rmiService); // Object will be used as a

// service by clients

} **catch** (RemoteException e) {

// Problem with starting the server

JOptionPane.*showMessageDialog*(**null**,

"Server can't start service. Please restart server.");

e.printStackTrace();

}

// Start server updater thread

serverUpdater = **new** ServerUpdater();

serverUpdater.start();

// Start code runner thread

codeRunnerThread = **new** CodeRunnerThread();

codeRunnerThread.start();

}

**4.1.2.3 CodeRunner thread from Contest Admin**

The job of this thread is to look in the database if there are any pending submissions, if there aren’t any pending submissions, then the thread will sleep for 30 seconds and it will repeat that there is one or more pending submissions. If there is pending submissions, then it will pass the submission to the CodeRunner class (explained in detail in section 4.1.2.6) which will run and evaluate the submissions and the result it gives will automatically be updated in the database.

/\*\*

\* The job of this thread is to check for pending submissions and compile

\* them.

\*/

**class** CodeRunnerThread **extends** Thread {

**public** **void** run() {

**while** (**true**) {

ArrayList<Submission> submissions = DbAdapter

.*getPendingSubmissions*();

// If no pending submissions

**if** (submissions.size() == 0) {

**try** {

// Then rest for half a minute

Thread.*sleep*(30000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

} **else** {

// Run and validate the first submission available and

// update DB.

DbAdapter.*updateSubmission*(CodeRunner

.*runAndValidate*(submissions.get(0)));

}

}

}

}

**4.1.2.4 Database Adapter code snippet**

The Database Adapter is where all the database operations occurs, so if any database operation occurs in the system, then there was a call to the DatabaseAdapter class. It was designed in such a way that when inserting and updating, it takes an object, and when fetching data it also returns an object rather than returning a vague delimiter separated string, and the reason it was done in this manner is because it made more sense and makes the code easier and better to understand. An example below shows snippet of how all the ‘Programming Languages’ are returned from the database, they are returned as an ArrayList of objects from the Language class. And it is a synchronized method because it won’t allow more than one thread to access the same method at the same time and it was done to prevent inconsistencies in the database.

/\*\*

\* This will return an ArrayList of all the languages in the database.

\*/

**public** **static** **synchronized** ArrayList<Language> getAllLanguages() {

Connection c = **null**;

Statement stmt = **null**;

ArrayList<Language> languages = **new** ArrayList<Language>();

**try** {

Class.*forName*("org.sqlite.JDBC");

c = DriverManager.*getConnection*(*CONNECTION\_STRING*);

c.setAutoCommit(**false**);

stmt = c.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM LANGUAGE;");

**while** (rs.next()) {

languages

.add(**new** Language(rs.getInt("ID"),

rs.getString("NAME"), rs

.getString("FILE\_EXTENSION"), rs

.getString("CMD\_COMPILE"), rs

.getString("CMD\_EXECUTE")));

}

rs.close();

stmt.close();

c.close();

} **catch** (Exception e) {

System.*err*.println(e.getClass().getName() + ": " + e.getMessage());

System.*exit*(0);

}

**return** languages;

}

**4.1.2.5 Configuration File serialization and deserialization snippet**

There is a class called the ConfigManager class, it holds settings such as whether the system is configured as Contestant or Contest Admin and other settings. But since objects are stored in the memory and there aren’t persisting since memory is volatile and data will be lost. So in order to save the data, Java Serialization was used in that class to store it the object state to a file and to retrieve it. Below is a code snippet showing how the object is being saved and loaded.

/\*\*

\* Serializes the current object into the configuration file.

\*/

**private** **void** saveToFile() {

**try** {

FileOutputStream fileOut = **new** FileOutputStream(*CONFIG\_FILE*);

ObjectOutputStream out = **new** ObjectOutputStream(fileOut);

out.writeObject(**this**);

out.close();

fileOut.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* Deserializes the configuration file and loads into the current object.

\*/

**private** **void** loadFromFile() {

**if** (!*configExists*())

**return**; // Exit if file doesn't exist

ConfigManager o = **null**;

**try** {

FileInputStream fileIn = **new** FileInputStream(*CONFIG\_FILE*);

ObjectInputStream in = **new** ObjectInputStream(fileIn);

o = (ConfigManager) in.readObject();

in.close();

fileIn.close();

} **catch** (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

**this**.setRole(o.getRole());

**this**.setServerIp(o.getServerIp());

}

**4.1.2.6 CodeRunner class**

This is the heart of the whole application, below is the complete CodeRunner.java source. Since it’s hard to understand by just reading, it’s best if it was explained. The first thing is it creates a temporary folder where all the work is done. In that folder, it creates the input file and it creates the user submission file which was submitted by the user, then it compiles it to see whether it generates the correct output file, and then it creates the correct output file and gets the MD5 hash of both of the files; the user-generated output, and the expected output file. It will test to see if the hashes match, if they do match then the user submission is accepted; otherwise it will display the appropriate error. The CodeRunner is smart enough to identify whether the submission is accepted, rejected, no output was produced, or whether there was a compilation error, or a runtime error. So basically, the class will take a submission object containing the user submission data, and after compiling and running, it will modify the submission object which was originally given by changing the result in the object state. The modified Submission object will later on be used in the DatabaseAdapter class to update the submission with the new result instead of “Pending”.

**package** util;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.security.MessageDigest;

**import** java.security.NoSuchAlgorithmException;

**import** model.Submission;

/\*\*

\* This class is in charge of taking the getting a pending submission, compile

\* it, run it, validate it. It will give the user submission the input file, and

\* then it will compare the user submission output with the expected output, and

\* it will return the result based on that.

\*/

**public** **final** **class** CodeRunner {

/\*\*

\* This is where the directory name where the code compilation and running

\* will take place.

\*/

**private** **static** **final** String *WORKING\_DIRECTORY* = "pcstemp";

/\*\*

\* Private constructor because it's a util class, no need for instantiation

\*/

**private** CodeRunner() {

}

/\*\*

\* Runs a command on the command line in the working directory.

\*/

**private** **static** **int** runProcess(String command) **throws** Exception {

Process process = Runtime.*getRuntime*().exec(command, **null**,

**new** File(*WORKING\_DIRECTORY*));

*printProcessLines*(command + " stdout:", process.getInputStream());

*printProcessLines*(command + " stderr:", process.getErrorStream());

process.waitFor();

**return** process.exitValue();

}

/\*\*

\* Prints texts from streams after running a command on the command line.

\*/

**private** **static** **void** printProcessLines(String name, InputStream ins)

**throws** Exception {

String line = **null**;

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(ins));

**while** ((line = in.readLine()) != **null**) {

System.*out*.println(name + " " + line);

}

}

/\*\*

\* Takes a submission and runs and validates it and returns the same object

\* with the modified output and result

\*/

**public** **synchronized** **static** Submission runAndValidate(Submission submission) {

// Re-usable objects

PrintWriter out = **null**;

String resultFileChecksum = **null**;

String outputFileChecksum = **null**;

**int** compileResult = -1;

**int** executeResult = -1;

// Create working directory

File dir = **new** File(*WORKING\_DIRECTORY*);

dir.mkdir();

// Create the input, output, and result, and submission file objects in

// the working

// directory

File inputFile = **new** File(dir.getAbsolutePath() + File.*separator*

+ "input.in");

File outputFile = **new** File(dir.getAbsolutePath() + File.*separator*

+ "output.out");

File resultFile = **new** File(dir.getAbsolutePath() + File.*separator*

+ "result.out");

File submissionFile = **new** File(dir.getAbsoluteFile() + File.*separator*

+ submission.getProblem().getFilename() + "."

+ submission.getLanguage().getFileExtension());

// Create input file only, output will be created later to prevent

// cheating

**try** {

inputFile.createNewFile();

out = **new** PrintWriter(inputFile);

out.print(submission.getProblem().getInput());

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

out.close();

}

// Create user submission file

**try** {

submissionFile.createNewFile();

out = **new** PrintWriter(submissionFile);

out.print(submission.getSourceCode());

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

out.close();

}

// Convert language variables to values

String cmdCompile = *replaceLanguageVars*(submission.getLanguage()

.getCmdCompile(), submission);

String cmdExecute = *replaceLanguageVars*(submission.getLanguage()

.getCmdExecute(), submission);

// Compile and execute program

**try** {

compileResult = *runProcess*(cmdCompile);

executeResult = *runProcess*(cmdExecute);

} **catch** (Exception e) {

e.printStackTrace();

}

// Create the results file

// It was created at this point to prevent cheating

**try** {

resultFile.createNewFile();

out = **new** PrintWriter(resultFile);

out.print(submission.getProblem().getOutput());

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

out.close();

}

// Store user output in submission object

**if** (outputFile.exists()) {

BufferedReader br;

StringBuffer sb = **new** StringBuffer();

String line = **null**;

**try** {

br = **new** BufferedReader(**new** FileReader(outputFile));

**while** ((line = br.readLine()) != **null**) {

sb.append(line).append("\n");

}

br.close();

submission.setOutput(sb.toString());

} **catch** (IOException e) {

e.printStackTrace();

}

// Get the checksum of the output file and result file

**try** {

resultFileChecksum = *getChecksum*(resultFile.toString());

outputFileChecksum = *getChecksum*(outputFile.toString());

} **catch** (NoSuchAlgorithmException | IOException e) {

e.printStackTrace();

}

}

// Determine result

**if** (compileResult != 0)

submission.setResult(Submission.*RESULT\_COMPILATION\_ERROR*);

**else** **if** (executeResult != 0)

submission.setResult(Submission.*RESULT\_RUNTIME\_ERROR*);

**else** {

**if** (outputFile.exists()) { // Check if output file was created

**if** (resultFileChecksum.equals(outputFileChecksum)) // Compare

// files

submission.setResult(Submission.*RESULT\_ACCEPTED*); // Return

// as

// accepted

**else**

submission.setResult(Submission.*RESULT\_REJECTED*); // Return

// as

// rejected

} **else**

submission.setResult(Submission.*RESULT\_NO\_OUTPUT*); // Return as

// no

// output

}

// Delete directory after use

Utilities.*deleteDirectory*(dir);

**return** submission;

}

/\*\*

\* Replace the language variables with the appropriate values

\*/

**private** **static** String replaceLanguageVars(String str, Submission s) {

**return** str.replaceAll("%NAME%", s.getProblem().getFilename());

}

/\*\*

\* This will create the checksum by getting the bytes from the file, it

\* doesn't need to be called It will be called automatically by the

\* getChecksum method.

\*/

**private** **static** **byte**[] createChecksum(String filename)

**throws** FileNotFoundException, NoSuchAlgorithmException, IOException {

InputStream fis = **new** FileInputStream(filename);

**byte**[] buffer = **new** **byte**[1024];

MessageDigest complete = MessageDigest.*getInstance*("MD5");

**int** numRead;

**do** {

numRead = fis.read(buffer);

**if** (numRead > 0) {

complete.update(buffer, 0, numRead);

}

} **while** (numRead != -1);

fis.close();

**return** complete.digest();

}

/\*\*

\* This function is the function that will be used to calculate the

\* checksum, pass a file, and it'll return the checksum.

\*/

**private** **static** String getChecksum(String filename)

**throws** FileNotFoundException, NoSuchAlgorithmException, IOException {

**byte**[] b = *createChecksum*(filename);

String result = "";

**for** (**int** i = 0; i < b.length; i++) {

result += Integer.*toString*((b[i] & 0xff) + 0x100, 16).substring(1);

}

**return** result;

}

}

**4.1.2.7 RMI Interface file**

The RMI interface file (Rmi.java) is a file that contains an Java interface and that interface will be used by both the client (Contestant) and the server (Contest Admin). This one just defines the methods that can be called. The server uses this interface so that it will implement them by providing the actual code and rebinding the object to the RMI registry so that it will be available to the clients. The clients use the interface so that it will know what the allowed method calls it can make are and what parameters that are required and the return types of every method call.

In brief there are two methods, the first one is the getData(String username, String password) method which takes a contestant username and password, and replies with an object of RmiData which has information from the server such as contest time, scoreboard, and other useful contest information if the username and password is correct, if it wasn’t correct, then the object will contain data regarding wrong username and password. The contestant frequently uses this at some time interval to fetch new data from the Contest Admin PC.

The other method is the sendSubmission(Submission submission, String username) method, and its job is that it takes an object from the Submission class containing data about the submission that is being sent and the username. This method returns a Boolean value, if it was true, then it means the server accepted the submission and its awaiting validation, otherwise the server rejected the submission and could be due to the fact that the contest isn’t running at that time, therefore it rejected the submission.

**package** rmi;

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**import** model.Submission;

/\*\*

\* This class is the RMI interface, this will be used by the client when

\* invoking the methods, and it will be used by the server as it implements

\* this.

\*/

**public** **interface** Rmi **extends** Remote {

**public** RmiData getData(String username, String password)

**throws** RemoteException;

**public** **boolean** sendSubmission(Submission submission, String username)

**throws** RemoteException;

}

**4.1.3 Validation Checks**

There are numerous validation checks in the Programming Contest System. The first example of validation checks used is the Configuration Screen that when configuring as a Contestant, the system won’t accept a fake Server IP address and will check to make sure it’s a valid IP address, and the “Configure” button will remain disabled until and only until the IP address supplied is a valid IP address as shown below:

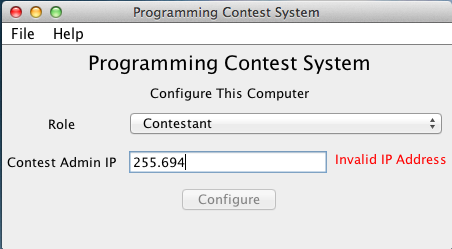


Figure 4.1. Validation – Configuration Screen IP Address

Login validation is another form of validation check which will check for the username and password provided, and will not allow login unless the credentials exists on the Contest Admin PC. If it doesn’t exist, then it will show an error as seen below:

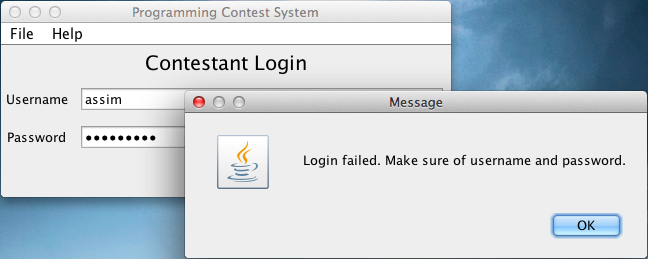


Figure 4.2. Validation – Contestant Login

In the server (Contest Admin PC), there’s a chance another process/application is listening on the port that the system is designed to listen from, or maybe there’s two or more instances of the server interface running which will lead to a conflict in port usage, in case that happens, it won’t be able to start and the error shown will as follows:

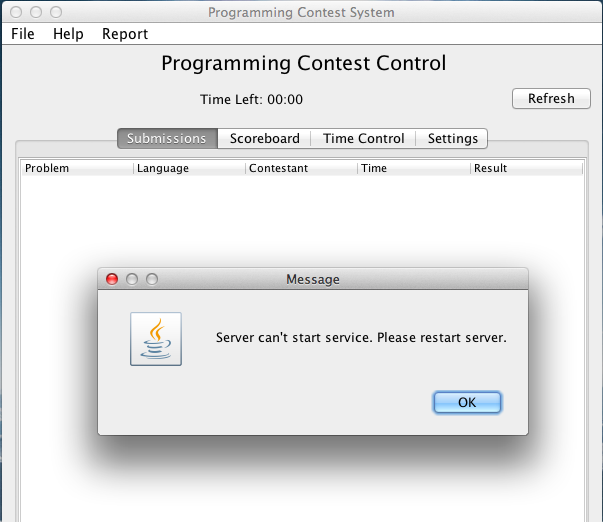


Figure 4.3. Validation – Server can’t start

There’s also a case when the Contest Admin PC software is offline and a contestant will try to login. The system will give a different type of error for this case, it will check whether the connection can be established with the Contest Admin PC or not, so the contestant can identify that the issue is not with the supplied username and password but it’s because a connection can’t be established due to either wrong IP address or the Contest Admin software isn’t running, or maybe any other network-related issue, then it will be showing the following:

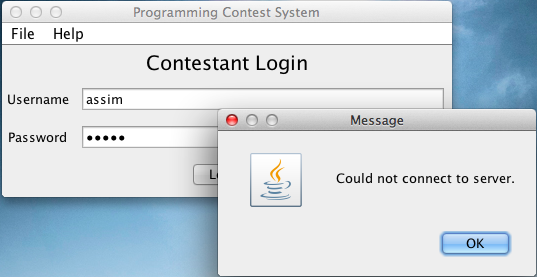


Figure 4.4. Validation – No connection to server

Another form of validation which is a very important one is that the Contest Admin Interface will block any submissions when contest is not running. If the contest isn’t running, the Contest Admin PC will reject submissions, and at the same time the Contestant screen will disable the “Submit Code” button when the contest isn’t running as shown below:

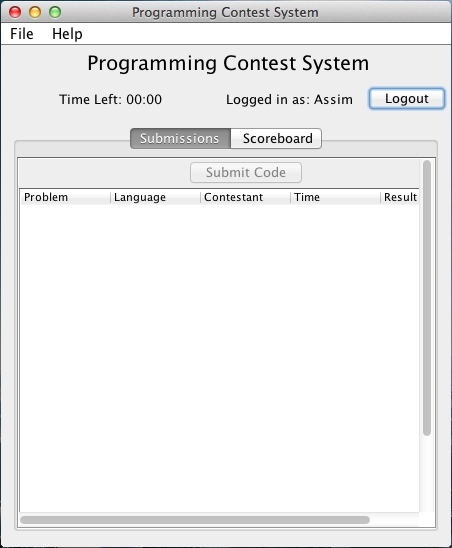


Figure 4.5. Validation – ‘Submit Code’ button disabling

Last but not least, the contest start time and end time range should be correct, so the Contest Admin can’t set the start time to be after the end time and vice-versa. The Contest Admin interface will automatically check and validate the input and if it was wrong, then it will show an error as in the screenshot below:

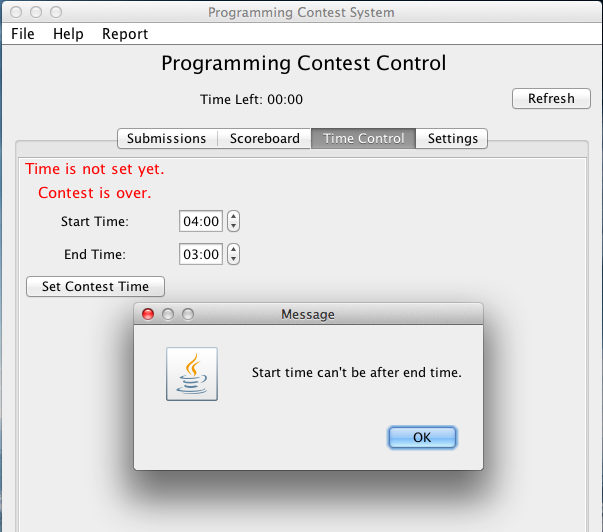


Figure 4.6. Validation – Contest time

**4.1.4 Reports**

There is just one major report that the system generates and that is the contest report. This is done by the Contest Admin by clicking on the “Report” menu from the menu bar and selecting the “Generate Report” option which will automatically generate a PDF file as shown below:

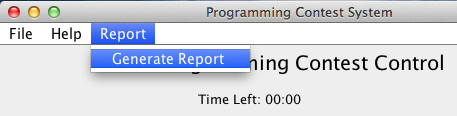


Figure 4.7. Generate Report from Menu

The PDF file that the system will generate will contain some statistics about the contest as well as the scoreboard as shown below:

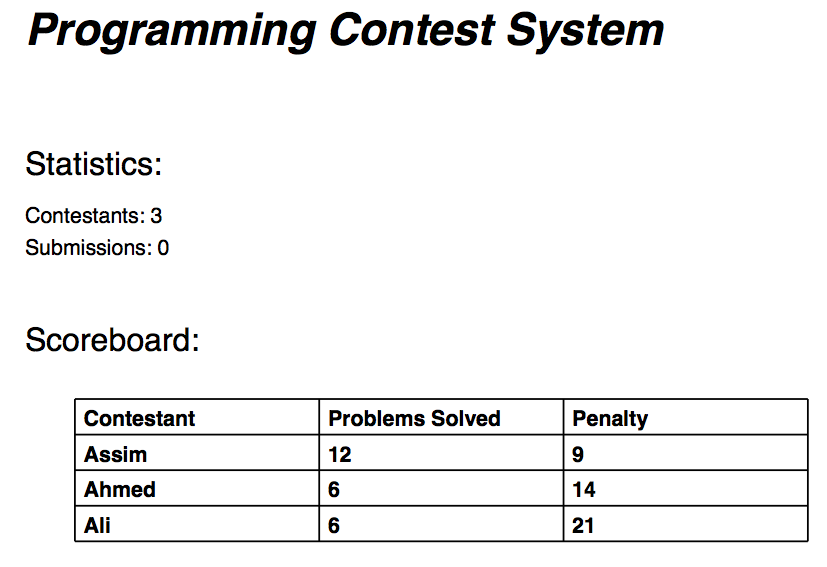


Figure 4.8. PDF Report Example

**4.2 Testing**

**White box Testing:**

The first type of testing that was done is the white box testing. This was testing different components by giving it a certain input and checking if that component generates the correct output. Even though there was a lot of testing done in this stage, the most important tests are listed down below:

* Check if data is being transferred correctly between machines.
* Check if submission is being compiled correctly.
* Check if Code Validator identifies submission result (Accepted, Runtime Error, Compilation Error, etc..) correctly.
* Make sure submissions are reached or throw an error to notify contestant to try to submit again.
* Prevent user submission when contest time is over even when contestant keeps submission form open for a long time.
* Make sure configuration screen only shows up if PC wasn’t configured or the configuration was reset.
* Make sure scoreboard shows the correct standings even when there’s a tie it should consider the penalties.
* Contestants should not be able to post submissions if the contest isn’t configured.
* The Contest should constantly update itself and fetch new data from Contest Admin.

**Black box Testing:**

After testing the internal components with the white box testing, it’s time to test the system components as a whole with black box testing. The technique used was choosing different configurations and different test cases and test and make sure it passed the test. So every test case will be tested multiple times and on different configurations. For each test mentioned in the table below, the software was configured in different configuration as listed:

* Contest Admin and 1 Contestant on the same PC. (1 PC)
* Contest Admin and 1 Contestant on the same PC, and another Contestant on another PC. (2 PCs total)
* Contest Admin on one PC, and one Contestant on another PC. (2 PCs)
* Contest Admin on one PC, and two Contestants on two different PCs. (3 PCs)

Now for every of the previous configurations listed above, the following test cases were performed and made sure that it passed on every configuration. The following are the list some of the most important black box test cases used for the project:

|  |  |
| --- | --- |
| **Test Case** | **Expected Output** |
| Contestant should be able to login, submit, and logout. | The submission should be added to the Contest Admin PC. |
| Contestant can’t post unless contest is running. | The Submit button should be disabled on the Contestant screen, and should not allow submissions unless the contest is running. |
| Contest Admin PC should continue to validate pending submissions even if it crashed with pending submissions after restarting. | Contest Admin interface should be closed with pending submissions, and when re-opening, it should continue validating those submissions. |
| Contest Admin PC should be able to accept multiple submissions at the same time from many contestants. | It should receive all of them and validate them without any issues. |

Table 4.1. Black Box Testing.

**Chapter 5**

**Conclusion and Future**

**5.1 Summary of Findings:**

Programming Contest System is about hosting programming contests by allowing them to send solutions to problems, and the judging and scoreboard generation happens on the fly based on the output generated from the contestant programs.

* The problems that were experienced are:
  + It was hard to track which contestants were connected or disconnected due to the issues in how network sockets are made.
  + Contestants and Contest Admin won’t be able to know whether they are connected or not due to the nature of TCP sockets.
  + It was hard to implement a mechanism to get updates from the Contest Admin PC in case of settings changed because of the nature of TCP sockets.
  + There used to be some problems where there’s a limit of bytes that could be sent in the network which led to incomplete submissions that led to wrong judging.
  + There were issues compiling and validating submissions using the Command Line Interpreter because on different machines due to different commands used on different operating systems (platform-independent).

**5.2 Conclusion**

During the execution stage of the system, time was limited and the project was very huge, so to overcome the problems mentioned. Most of problems are related to TCP sockets. To overcome the first three problems which are related to sockets, the whole system architecture had to be changed in such a way that the Contestant Interface requests for the latest information from the Contest Admin PC at some time interval, and this will help keep the connection alive as well as update the Contestant Interface in case of any updates in the Contest Admin Interface, and at the same time the client can know if the Contest Admin PC went offline. The Contest Admin will also be waiting from Contestants to send their submissions so that it will validate it. This way it solved nearly most of the TCP socket issues that were stopping the Programming Contest System from being completed.

Regarding the problem of the incomplete submissions data sent from Contestant PC to Contest Admin PC, which was due to how TCP sockets handling code in Java, most of the network specific code had to be handled which was a very tedious task, so to overcome this, Remote Method Invocation (RMI) was used. RMI allowed the Contestant to call objects in the Contest Admin PC which allowed transfer of data since it internally deals with sockets and serialization. Another reason for switching for normal TCP sockets to RMI because it allows moving objects from one machine to another and not only bytes of data, this helped build a much powerful system without re-inventing the wheel. Experienced Software Engineers would say “Don’t re-invent the wheel, unless you plan on learning more about wheels.”

The last and final problem that was solved was there were issues in compilation and validating the code that was submitted by the Contestant. First, the idea was to have the system generate batch scripts (.bat) and shell scripts (.sh) files depending on the operating system for Windows and Unix/Mac/Linux respectively. But that would lead to bugs since it had to check on which operating system it is being run, then different operating system versions/distributions might have different types of supported scripts, and when running these scripts to compile, there isn’t a callback to notify the system when compilation and running is done, and it would also be hard to identify submission errors such as ‘Runtime Errors’, ‘Compilation Errors’, and other errors. To solve this problem, for each submission, a temporary folder is made, and a Command Line Interpreter instance is called using Java Runtime class, and it will directly run the compilation and execution commands based on the contest settings. So this way, the Contest Admin can compile and validate programs on any operating system not only on Windows making it platform-independent.

**5.3 Future Scope**

There are some features that could be useful features for the future such as allowing the Contestants to write solutions that not only read from input file and output the result to output file, but also add an option to read and write directly in the Standard Streams. Standard Streams are I/O channels between a program and the text terminal, and there are connections such as standard input (stdin) for reading and standard output (stdout) for writing. So there’s a plan in allowing an option to read from stdin and generate output in stdout.

Apart from that, there is hope to have this system used for major programming contests such as the International Collegiate Programming Contest (ICPC) and other contests.

**References**

[1] Kathy Sierra and Bert Bates, Head First Java, O'Reilly Media, USA, February 2005

[2] Eric Freeman, Head First Design Patterns, O'Reilly Media, USA, Nov 2004

[3] Qusay H. Mahmoud, Sockets programming in Java: A tutorial, JavaWorld, USA, Nov 1996, http://www.javaworld.com/jw-12-1996/jw-12-sockets.html

[4] Java remote method invocation – Wikipedia, Retrieved March 2014. http://en.wikipedia.org/wiki/Java\_remote\_method\_invocation

[5] Paul Deitel and Harvey Deitel, Java: How to Program, 9th Edition, Prentice Hall, USA, March 2011