**ASSIGNMENT 1 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** |  | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** |  | **Student ID** |  |
| **Class** |  | **Assessor name** |  |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
|  |  | **Student’s signature** |  |

**Grading grid**

|  |
| --- |
| Grade (0-10) |
|  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **IV Signature:** | | |

Table of Contents

[I. Introduction 5](#_Toc106953895)

[II. Requirements 5](#_Toc106953896)

[III. UI design 6](#_Toc106953897)

[1. Login 6](#_Toc106953898)

[2. Registering 7](#_Toc106953899)

[3. Student list 8](#_Toc106953900)

[4. Add student 9](#_Toc106953901)

[5. Edit student 10](#_Toc106953902)

[6. View student information 11](#_Toc106953903)

[IV. Implementation 11](#_Toc106953904)

[1. Program structure 11](#_Toc106953905)

[2. Classes 11](#_Toc106953906)

[2.1. PersonModel 11](#_Toc106953907)

[2.2. StudentModel 12](#_Toc106953908)

[2.3. UserModel 13](#_Toc106953909)

[2.4. LoginController 13](#_Toc106953910)

[2.5. RegisterController 13](#_Toc106953911)

[2.6. StudentController 13](#_Toc106953912)

[3. important algorithms 14](#_Toc106953913)

[3.1. calculateGPA 14](#_Toc106953914)

[3.2. ReadUserFromFile 14](#_Toc106953915)

[3.3. UserLogin 15](#_Toc106953916)

[3.4. WriteToFile 16](#_Toc106953917)

[3.5. CheckIfExist 17](#_Toc106953918)

[3.6. FindStudentByID 18](#_Toc106953919)

[3.7. FindStudentByName 18](#_Toc106953920)

[3.8. DeleteStudent 19](#_Toc106953921)

[3.9. DeleteAllStudent 20](#_Toc106953922)

[3.10. SortStudentByGPA 20](#_Toc106953923)

[3.11. populateTable 22](#_Toc106953924)

[4. error handling 23](#_Toc106953925)

[4.1. Checking email address 23](#_Toc106953926)

[4.2. Checking phone number 23](#_Toc106953927)

[4.3. Check date 23](#_Toc106953928)

[4.4. Check grade 24](#_Toc106953929)

[V. Test 25](#_Toc106953930)

[1. Test plan 25](#_Toc106953931)

[2. JUnit test 30](#_Toc106953932)

[2.1. testRegisterStudent 32](#_Toc106953933)

[2.2. testGetStudentList 33](#_Toc106953934)

[2.3. testCheckEmail 33](#_Toc106953935)

[2.4. testCheckDate 34](#_Toc106953936)

[2.5. testCheckPhone 35](#_Toc106953937)

[2.6. testCheckStudent 35](#_Toc106953938)

[2.7. testReadStudentFromFile 36](#_Toc106953939)

[2.8. testDeleteStudent 37](#_Toc106953940)

[2.9. testDeleteAllStudent 37](#_Toc106953941)

[2.10. testSortStudentByGpaAscending 38](#_Toc106953942)

[2.11. testSortStudentByGpaDescending 39](#_Toc106953943)

[VI. Result 40](#_Toc106953944)

[VII. Conclusion 58](#_Toc106953945)

[References 59](#_Toc106953946)

# Introduction

The staff from Doan Ket secondary school is in need of a program that allows them to conveniently manage students’ information. For that, they have reached out to FPT Software, where the author is currently employed as a software engineer, to request for a small software that has functions where students’ information can be modified and updated easily and intuitively.

# Requirements

# UI design

## Login

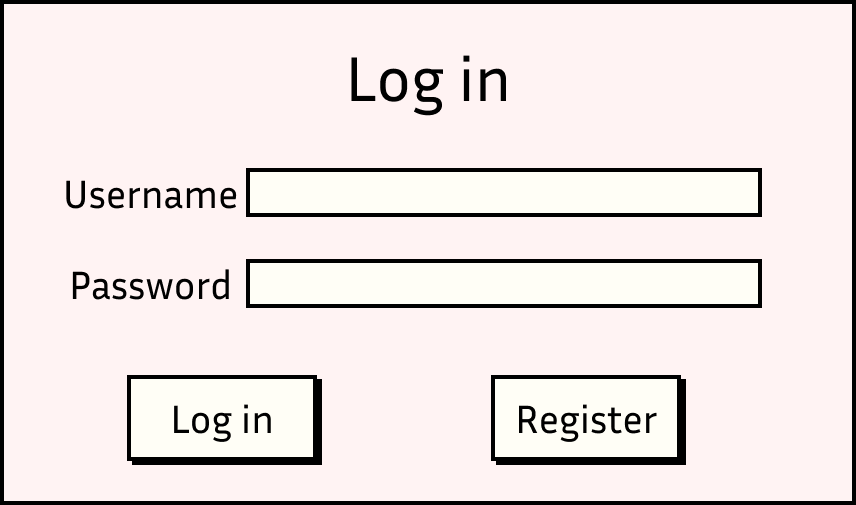


Figure 1 Login screen

## Registering

Graphical user interface, application

Description automatically generated with medium confidence

Figure 2 Register window

## Student list

Graphical user interface

Description automatically generated

Figure 3 Index window

## Add student

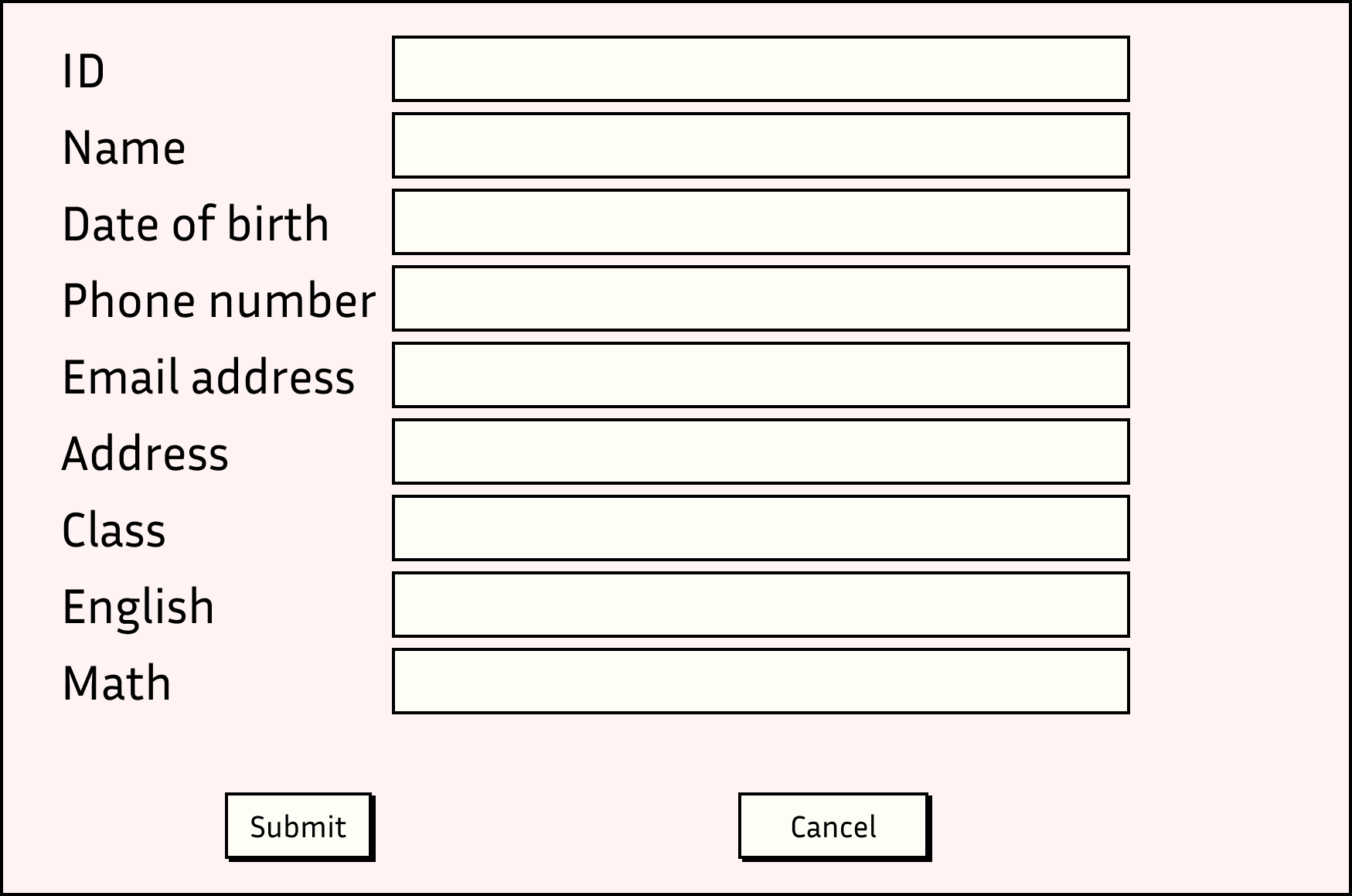


Figure 4 Window to add new student

## Edit student

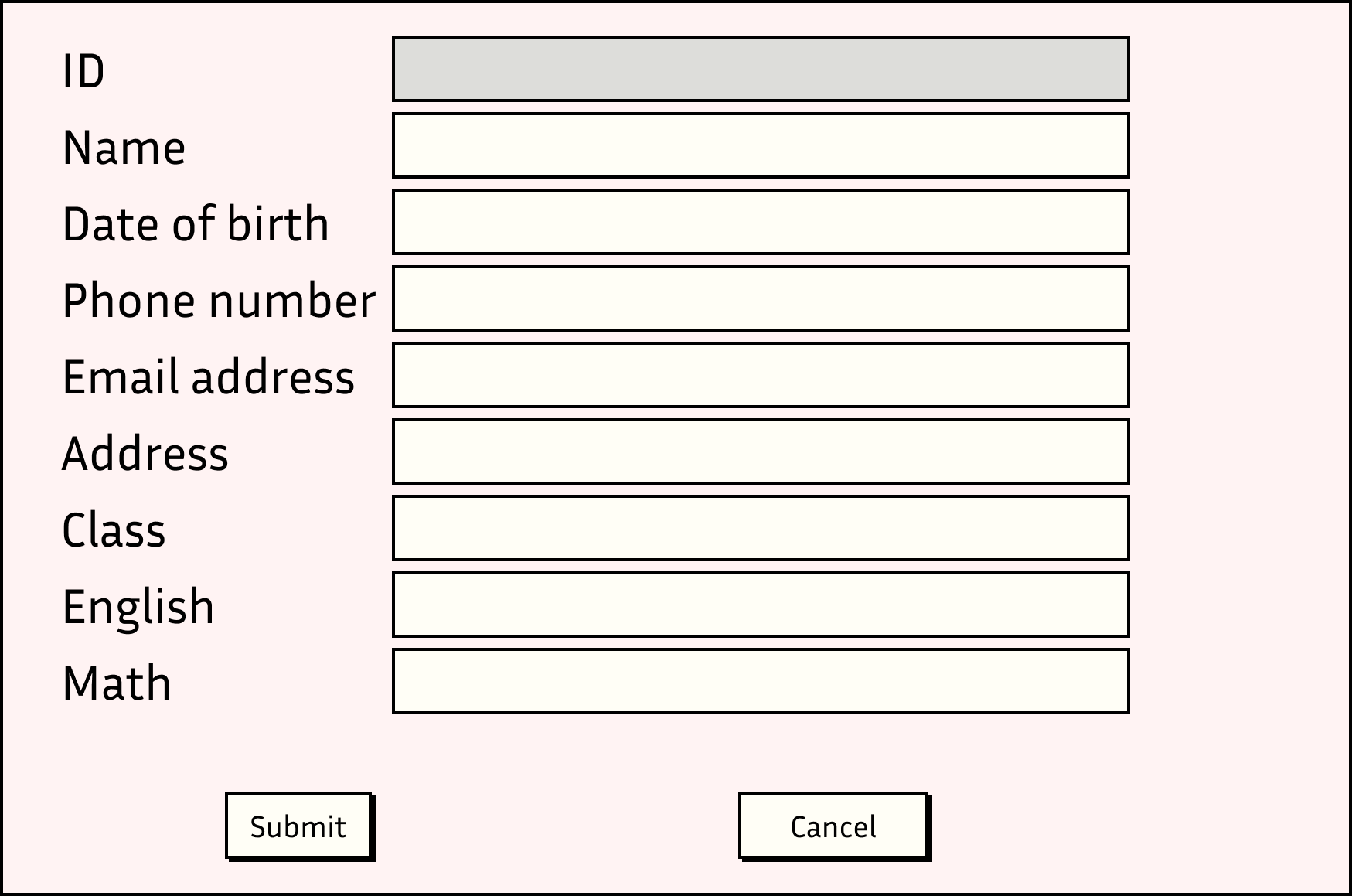


Figure 5 Edit a student's information

## View student information

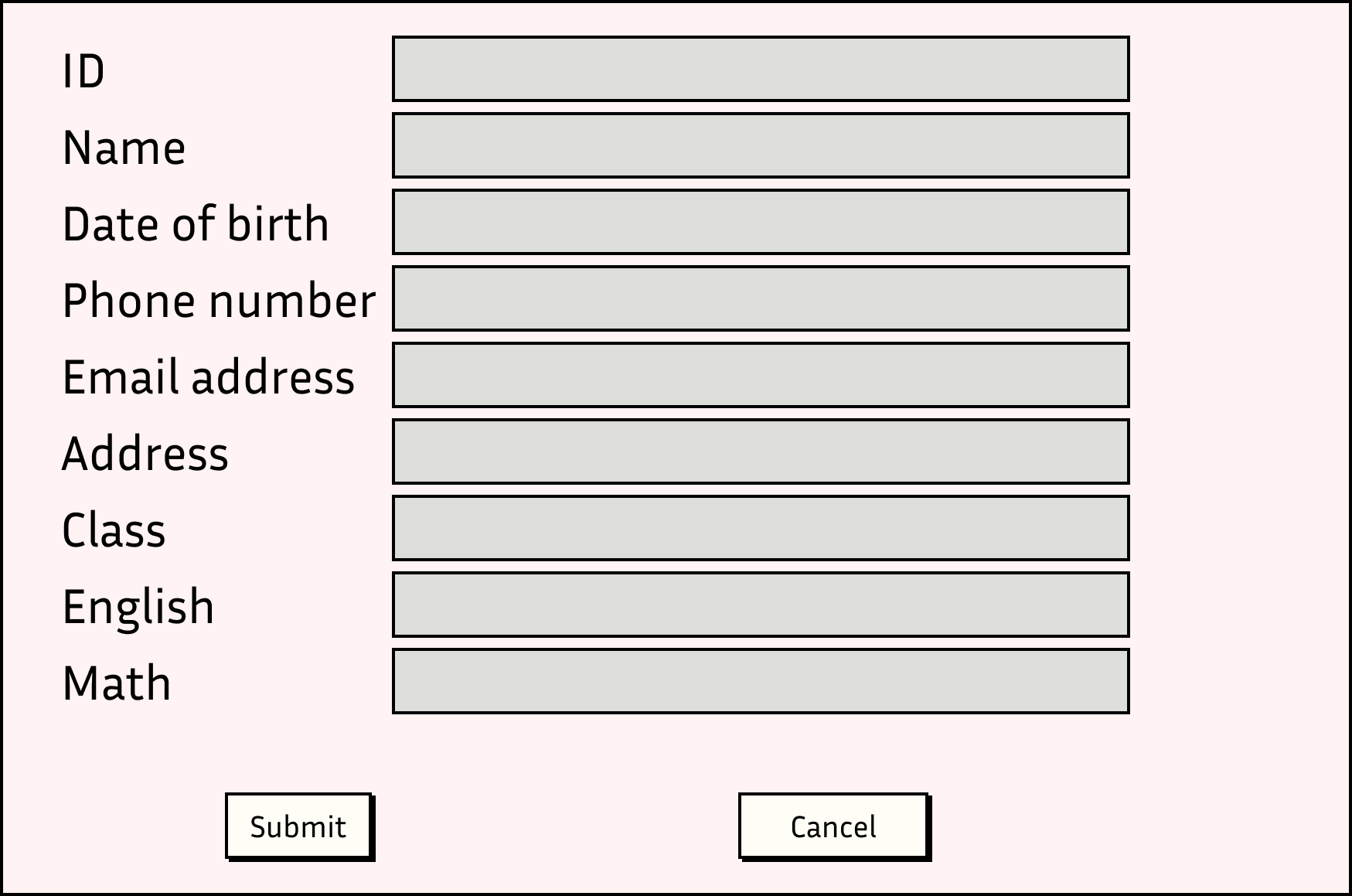


Figure 6 View a student's information

# Implementation

## Program structure

This program uses the MVC model which divides the code into three different components called Model, View and Controller. Each components handles a specific side of the program. (Tutorialspoint, 2022)

In this program, the Model component is used to store the structure of every object data type. The Controller the uses it to create instances of the objects and then store them in a file.

The View component is responsible for the application's UI logic. It presents the user with forms, fields, and table for them to interact with the program intuitively.

## Classes

### PersonModel

StudentModel holds the components of necessary to create a person object and modify it. Having a Person model is optimal because should the program expand and implement the feature to manage teachers’ information, the development would take much less time and efforts

|  |  |
| --- | --- |
| Variable | Type |
| ID | string |
| Name | string |
| Email | string |
| Phone number | string |
| Address | string |
| Date of birth | string |

Table 1 Variables of PersonModel

|  |  |
| --- | --- |
| Method | Description |
| Constructor | To create a new person with all the variables included within the model |
| Empty constructor | To create a new person with no information and then use setters to set each information |
| Getters | Gets the information of the instance and returns it |
| Setters | Sets the information of the instance |

Table 2 Methods of PersonModel

### StudentModel

StudentModel holds the components of necessary to create a student object and modify it.

|  |  |
| --- | --- |
| Variable | Type |
| mathGrade | int |
| englishGrade | int |
| gpa | double |
| classID | String |

Table 3 Variables of StudentModel

|  |  |
| --- | --- |
| Method | Description |
| Constructor | To create a new person with all the variables included within the model |
| Empty constructor | To create a new person with no information and then use setters to set each information |
| Getters | Gets the information of the instance and returns it |
| Setters | Sets the information of the instance |
| calculateGPA | This function is to calculate the GPA of the student by dividing the sum of Math and English grade of the student |
| getStudentInfo | Print the student's information out in JSON format |
| getStudentInfo2 | Print the student's information out in CSV format |

Table 4 Methods of StudentModel

### UserModel

UserModel is a model used to hold the information of every user’s login information including username and password.

|  |  |
| --- | --- |
| Variable | Type |
| username | string |
| password | string |

Table 5 Variables of UserModel

|  |  |
| --- | --- |
| Method | Description |
| Constructor | To create a new person with all the variables included within the model |
| Empty constructor | To create a new person with no information and then use setters to set each information |
| Getters | Gets the information of the instance and returns it |
| Setters | Sets the information of the instance |
| getUserInfo | Print the user information out in JSON format |
| getUserInfo2 | Print the user information out in CSV format |

Table 6 Methods of UserModel

### LoginController

This Controller is used to control events regarding logins. When a login is performed, the program reads the file containing a list of users, parses them into an ArrayList of User objects before trying to find a match with the login input by the user. After the login is performed, a message displaying its status will pop up and lead the user to the appropriate window. When the user click on the “Register” button, they are taken to the register window.

### RegisterController

This Controller is used to control events regarding registration. When a registration is performed, the program reads the file containing a list of users, parses them into an ArrayList of User objects before trying to find a match with the registration input by the user. If there is a match with the username, an error message is displayed and asks the user to enter a different username. If there is no matching username, a success message is displayed, and the user is taken to the login window.

### StudentController

This Controller is used to control events regarding student information. It is responsible for writing student information onto a file. It is also responsible for reading student information from a file and putting them into an ArrayList of Student objects. Another important part of this controller is that it handles the data validation and error handling when the user tries to submit a student’s information.

## important algorithms

### calculateGPA

When a new Student is created or edited using a constructor, the program automatically calculates the GPA of that student based on the English and Math grade of that Student input by the user Text

Description automatically generated

Figure 7 GPA is automatically calculated

GPA is calculated by averaging the sum of Math and English grade and parse it as a double data type

public double calculateGpa() {

        double gpa = (double) (getMathGrade() + getEnglishGrade()) / 2;

        return gpa;

    }

### ReadUserFromFile

This algorithm takes in a String as the name of the file and an ArrayList of User so it can write into it. Then it declares the variable line for each User and delimiter to know how each column is separated. Then it uses BufferedReader to read each line of the file, put the information into a new Student object and put that object into the ArrayList.

public static void readUserFromFile(String fileName, ArrayList<UserModel> userList) {

        String line = "";

        String cvsSplitBy = ",";

        try {

            // Read from file user.csv

            java.io.FileReader fileReader = new java.io.FileReader(fileName);

            java.io.BufferedReader bufferedReader = new java.io.BufferedReader(fileReader);

            while ((line = bufferedReader.readLine()) != null) {

                // Split the line by comma

                String[] userInfo = line.split(cvsSplitBy);

                // Create a new user object

                UserModel user = new UserModel(userInfo[0], userInfo[1]);

                // Add the user object to the userList

                userList.add(user);

            }

            bufferedReader.close();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

### UserLogin

After the user has input the login credentials and click on the “Login” button, the program read the file for a list of Users and compare each of them with the input made by the user. If there is a match, the program displays a success message and takes the user to the index window. Otherwise, an error message notifying the user of wrong login credentials and ask them to re-enter the information.

// Read from file user.csv and create a new user object for each line

    // user.csv

    public static void loginUser(String userName, String userPassword) {

        String fileName = "user.csv";

        readUserFromFile(fileName, userList);

        isLogin = false;

        for (UserModel user : userList) {

            if (user.getUserName().equals(userName) && user.getUserPassword().equals(userPassword)) {

                isLogin = true;

                break;

            }

        }

        if (isLogin) {

            JOptionPane.showMessageDialog(null, "Login successfully");

            // go to StudenListGUI

            StudentListGUI studentListGUI = new StudentListGUI();

            studentListGUI.setVisible(true);

        } else {

            JOptionPane.showMessageDialog(null, "Login failed");

        }

    }

### WriteToFile

When the user performs an account or student registration, the program writes their data onto a file, if the file does not yet exist, the program creates a file for it to be written on. It uses FileWriter to write each User information on to a line and end each User information with a “\n” meaning a line break.

// write to file

    public static void writeUserToFile(String fileName, String userInfo2) {

        try {

            File file = new File(fileName);

            if (!file.exists()) {

                file.createNewFile();

            }

            // write to file

            FileWriter fw = new FileWriter(file, true);

            fw.write(userInfo2 + "\n");

            fw.close();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

### CheckIfExist

When a user registers for a new User, the program checks if the username already exists on the system. It does this by reading the File for Users and try to find a match with the input by the user. If there is a match, it returns a boolean depicting the status of the finding.

// Check if userName is already exist

    public static boolean checkUserName(String userName) {

        boolean isExist = false;

        String fileName = "user.csv";

        readUserFromFile(fileName, userList);

        for (UserModel user : userList) {

            if (user.getUserName().equals(userName)) {

                isExist = true;

                break;

            }

        }

        return isExist;

    }

Similar algorithms is applied for checking existing Student when registering for a new one

// check if the student is already exist

    public static boolean checkStudent(String id) {

        boolean isExist = false;

        String fileName = "student.csv";

        readStudentFromFile(fileName, studentList);

        for (StudentModel student : studentList) {

            if (student.getId().equals(id)) {

                isExist = true;

            }

        }

        return isExist;

    }

### FindStudentByID

This feature takes in a String as the ID needed to be searched for. It reads the file for a list of Students and go through the list to find the Student with matching id before returning that Student.

// Find student by id

    public static StudentModel findStudent(String id) {

        StudentModel student = null;

        String fileName = "student.csv";

        // clear studentList

        studentList.clear();

        readStudentFromFile(fileName, studentList);

        for (StudentModel s : studentList) {

            if (s.getId().equals(id)) {

                student = s;

            }

        }

        return student;

    }

### FindStudentByName

This operates similarly to FindStudentByID, but instead of returning just one student, it return a list of students that have name containing the input characters.

// Find student by name and return a list of students

    public static ArrayList<StudentModel> findStudentByName(String name) {

        ArrayList<StudentModel> studentList = new ArrayList<StudentModel>();

        String fileName = "student.csv";

        readStudentFromFile(fileName, studentList);

        ArrayList<StudentModel> studentList2 = new ArrayList<StudentModel>();

        for (StudentModel student : studentList) {

            if (student.getName().contains(name)) {

                studentList2.add(student);

            }

        }

        return studentList2;

    }

### DeleteStudent

This feature takes in the ID of the student needed to delete, find the Student with that ID and deletes it from the ArrayList. It then clears all the data in the saved file and replace it with the updated list

// Find student id and delete it

    public static void deleteStudent(String id) {

        StudentModel student = findStudent(id);

        if (student != null) {

            studentList.remove(student);

            String studentInfo2 = student.getStudentInfo2();

            String fileName = "student.csv";

            deleteAllStudent();

            for (StudentModel s : studentList) {

                writeStudentToFile(fileName, s.getStudentInfo2());

            }

        } else {

            JOptionPane.showMessageDialog(null, "Student not found");

        }

    }

### DeleteAllStudent

This feature deletes all students in the file. It works by opening up the student file and replace everything with a single space “ “.

// Delete data on the file and replace with the new studentList

    public static void deleteAllStudent() {

        String fileName = "student.csv";

        try {

            File file = new File(fileName);

            if (!file.exists()) {

                file.createNewFile();

            }

            FileWriter fw = new FileWriter(file, false);

            fw.write("");

            fw.close();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

### SortStudentByGPA

This algorithms first clears all the elements in the studentList and get a new studentList by reading the file to avoid data duplication. It then uses comparator to sort the list by GPA in ascending or descending order

// Sort the student list by gpa ascending

    public static void sortStudentByGpaAscending() {

        studentList.clear();

        studentList = getStudentList();

        Collections.sort(studentList, new Comparator<StudentModel>() {

            @Override

            public int compare(StudentModel o1, StudentModel o2) {

                // round up the return

                return (int) Math.round(o1.getGpa() - o2.getGpa());

            }

        });

        // studentList.clear();

    }

    // Sort the student list by gpa descending

    public static void sortStudentByGpaDescending() {

        studentList.clear();

        studentList = getStudentList();

        Collections.sort(studentList, new Comparator<StudentModel>() {

            @Override

            public int compare(StudentModel o1, StudentModel o2) {

                // round up the return

                return (int) Math.round(o2.getGpa() - o1.getGpa());

            }

        });

        // studentList.clear();

    }

### populateTable

After the program has read and put the information into an ArrayList, the table needs to be populated with the list data to display it to the user. It takes in the table needed to populate, goes through each of the Student and fill each row with the information of each Student.

// populate table, read from studentList and write to table with their id, name,

    // class, math, eng, gpa

    public static void populateTable(JTable table) {

        // clear table

        table.setModel(new DefaultTableModel());

        // populate table

        for (StudentModel student : studentList) {

            String[] row = { student.getId(), student.getName(), student.getClassId(), student.getMathGrade() + "",

                    student.getEnglishGrade() + "", student.getGpa() + "" };

            ((DefaultTableModel) table.getModel()).addRow(row);

        }

        // if number of rows in the table

        if (table.getRowCount() > 0) {

            isNull = false;

        } else {

            isNull = true;

        }

    }

## error handling

### Checking email address

The conventional email format is [name@domain.domain](mailto:name@domain.domain). To ensure that the user enter the correct email format in order for the student to be registered into the system. To do this, regex is utilized so that the program can check for the formatting of the input.

// check the email format

    public static boolean checkEmail(String email) {

        boolean isValid = false;

        String emailRegex = "^[a-zA-Z0-9\_+&\*-]+(?:\\." + "[a-zA-Z0-9\_+&\*-]+)\*@"

                + "(?:[a-zA-Z0-9-]+\\.)+[a-z" + "A-Z]{2,7}$";

        if (email.matches(emailRegex)) {

            isValid = true;

        }

        return isValid;

    }

### Checking phone number

The phone number conventional format is 10 digits of integer from 0 to 9, the user has to input the correct format to add a new student into the system.

// Check phone number format

    public static boolean checkPhone(String phone) {

        boolean isValid = false;

        String phoneRegex = "^[0-9]{10}$";

        if (phone.matches(phoneRegex)) {

            isValid = true;

        }

        return isValid;

    }

### Check date

The conventional date format is dd/mm/yyyy, to make sure that the user input the correct format before a student is added into the system, a date checker using regex is implemented.

// check the date format

    public static boolean checkDate(String date) {

        boolean isValid = false;

        String dateRegex = "^(?:(?:31(\\/|-|\\.)(?:0?[13578]|1[02]))\\1|(?:(?:29|30)(\\/|-|\\.)"

                + "(?:0?[1,3-9]|1[0-2])\\2))(?:(?:1[6-9]|[2-9]\\d)?\\d{2})$|"

                + "(?:29(\\/|-|\\.)0?2\\3(?:(?:(?:1[6-9]|[2-9]\\d)?(?:0[48]|[2468][048]|[13579][26])|"

                + "(?:(?:16|[2468][048]|[3579][26])00))))$|"

                + "(?:0?[1-9]|1\\d|2[0-8])(\\/|-|\\.)(?:(?:0?[1-9])|(?:1[0-2]))\\4(?:(?:1[6-9]|[2-9]\\d)?\\d{2})$";

        if (date.matches(dateRegex)) {

            isValid = true;

        }

        return isValid;

    }

### Check grade

The grading format for this program is an integer from 0-100 inclusive, methods have been implemented to make sure that the user enters the right format if they want to add a new student into the system. First the program reads the input to see if it is an integer, if not, the user needs to re-enter, if yes, the program checks if the integer is in the range of 0-100 before letting the Student be added into the program.

 try {

                        Integer.parseInt(engGradeField.getText());

                        Integer.parseInt(mathGradeField.getText());

                } catch (NumberFormatException e) {

                        JOptionPane.showMessageDialog(this, "Please enter valid grades", "Error",

                                        JOptionPane.ERROR\_MESSAGE);

                        return;

                }

// check grade if it is valid

    public static boolean checkGrade(int grade) {

        boolean isValid = false;

        if (grade >= 0 && grade <= 100) {

            isValid = true;

        }

        return isValid;

    }

# Test

## Test plan

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Description | Data | Expected result | Actual result | Status |
| 1 | Register with valid data | username: admin password: admin | A success message display, user is returned to login screen, and "admin,admin" is added into user.csv | As expected | Pass |
|  | Register with invalid data by input the same information as last step | username: admin password: admin | A fail message displays, no new data is written on the file user.csv | As expected | Pass |
| 2 | Login with valid data | username: admin password: admin | A success message display, user is directed to student index window | As expected | Pass |
|  | Login with invalid data | username: asasasasasadmin password: admin | A fail message display, user stays on the login window | As expected | Pass |
| 3 | Add a student with valid data | ID:1 name: A date of birth: 12/12/2000 phone number: 1234567890 email address: A@gmail.com address: A street class: 12D english: 40 math: 50 | A success message display, user is directed to student index window, the new student is added into the table and student.csv | As expected | Pass |
|  | Add a student with duplicated ID | ID:1 name: B date of birth: 12/12/2000 phone number: 1234567890 email address: B@gmail.com address: A street class: 10A english: 90 math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Add a student with wrong email format | ID:12 name: B date of birth: 12/12/2000 phone number: 1234567890 email address: Bgmailcom address: A street class: 10A english: 90 math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Add a student with wrong phone number format | ID:12 name: B date of birth: 12/12/2000 phone number: 1234asffe email address: B@gmail.com address: A street class: 10A english: 90 math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Add a student with wrong date format | ID:12 name: B date of birth: 12122000 phone number: 1234567890 email address: B@gmail.com address: A street class: 10A english: 90 math: 32 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Add a student with wrong grade format | ID:12 name: B date of birth: 12/12/2000 phone number: 1234567890 email address: B@gmail.com address: A street class: 10A english: 90e math: 32 | A failure message display, no data is added to table or student.csv | As expected | Pass |
|  | ID:12 name: B date of birth: 12/12/2000 phone number: 1234567890 email address: B@gmail.com address: A street class: 10A english: 90 math: 32e | A failure message display, no data is added to table or student.csv | As expected | Pass |
| 4 | Edit a student with valid data | ID:1 name: A edited date of birth: 12/12/2000 phone number: 1234567890 email address: A@gmail.com address: A street class: 12D english: 40 math: 50 | Success message display, the user is directed back to the index menu and the data of student whose ID is 1 is updated | As expected | Pass |
|  | Edit a student with wrong email format | ID:1 name: A edited date of birth: 12/12/2000 phone number: 1234567890 email address: Agmailcom address: A street class: 12D english: 40 math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Edit a student with wrong phone number format | ID:1 name: A edited date of birth: 12/12/2000 phone number: 12345asd email address: A@gmail.com address: A street class: 12D english: 40 math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Edit a student with wrong date format | ID:1 name: A edited date of birth: 12122000 phone number: 1234567890 email address: A@gmail.com address: A street class: 12D english: 40 math: 50 | A failure message display, user is directed to student index window, no data is added to table or student.csv | As expected | Pass |
|  | Edit a student with wrong grade format | ID:1 name: A edited date of birth: 12/12/2000 phone number: 1234567890 email address: A@gmail.com address: A street class: 12D english: 40e math: 50 | A failure message display, no data is added to table or student.csv | As expected | Pass |
|  | ID:1 name: A edited date of birth: 12/12/2000 phone number: 1234567890 email address: A@gmail.com address: A street class: 12D english: 40 math: 50e | A failure message display, no data is added to table or student.csv | As expected | Pass |
| 5 | Search for student by name | name: A | All students whose name contains "A" is displayed on the table | As expected | Pass |
| 6 | Search for students by ID | ID:1 | The student whose ID is 1 is displayed | As expected | Pass |
| 7 | Delete all student |  | All students are deleted from student.csv | As expected | Pass |
| 8 | Delete student | ID: 1 | The student whose ID is 1 is deleted | As expected | Pass |

Table 7 Test plan

## JUnit test

To make the testing done more easily and conveniently, Junit has been implemented to perform an array of multiple tests simultaneously.

To aid the test, some generator has been programmed to generate random data for each column.

// random string generator

    public String randomString(int length) {

        StringBuilder sb = new StringBuilder();

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

            sb.append((char) ((int) (Math.random() \* 26) + 97));

        }

        return sb.toString();

    }

    // random email generator

    public String randomEmail(int length) {

        StringBuilder sb = new StringBuilder();

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

            sb.append((char) ((int) (Math.random() \* 26) + 97));

        }

        return sb.toString() + "@gmail.com";

    }

    // random phone number generator that consists of all integer

    public String randomPhone(int length) {

        StringBuilder sb = new StringBuilder();

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

            sb.append((int) (Math.random() \* 10));

        }

        return sb.toString();

    }

    // random date of birth generator, date, month, year are separated by "/"

    public String randomDob() {

        StringBuilder sb = new StringBuilder();

        int date = (int) (Math.random() \* 30) + 1;

        int month = (int) (Math.random() \* 12) + 1;

        int year = (int) (Math.random() \* 100) + 1900;

        sb.append(date);

        sb.append("/");

        sb.append(month);

        sb.append("/");

        sb.append(year);

        return sb.toString();

    }

    // random grade generator

    public int randomGrade() {

        return (int) (Math.random() \* 100) + 1;

    }

### testRegisterStudent

The first implemented test is testRegisterStudent which constructs a new Student object. When a student is successfully registered, the Controller assigns the value “true” to the variable “isRegister”. The test read this variable and determine if the test passes.

@Test

    public void testRegisterStudent() {

        System.out.println("registerStudent");

        // if a message dialog saying "Register successfully" is shown, then this test

        // is passed

        // student information include name, id, email, phone number, address, dob,

        // mathGrade, englishGrade, classId

        String name = randomString(20);

        String id = randomString(20);

        String email = randomEmail(20);

        String phone = randomPhone(10);

        String address = randomString(20);

        String dob = randomDob();

        int mathGrade = randomGrade();

        int englishGrade = randomGrade();

        String classId = randomString(20);

        studentController instance = new studentController();

        boolean expResult = true;

        instance.registerStudent(name, id, email, phone, address, dob, mathGrade, englishGrade, classId);

        boolean result = instance.isRegister;

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testGetStudentList

The second test is the test of the feature that reads a file and pass the data into a list of Students. The test then read the list, if the list is not empty, then it passes.

@Test

    public void testGetStudentList() {

        System.out.println("getStudentList");

        studentController instance = new studentController();

        // student list is returned

        // ArrayList<Student> expResult = null;

        studentController.studentList.clear();

        ArrayList<StudentModel> result = null;

        instance.getStudentList();

        // if result is not null, then this test is passed

        assertNotNull(studentController.studentList);

        // TODO review the generated test code and remove the default call to fail.

        // fail("The test case is a prototype.");

    }

### testCheckEmail

The next test is to see if the checkEmail function can properly validate an email formatted String. The test generate a random String followed by “@gmail.com”. If the String follows the conventional email format of [email@domain.domain2](mailto:email@domain.domain2), then it returns a boolean value “true” and the test passes.

@Test

    public void testCheckEmail() {

        // check email with a random email

        System.out.println("checkEmail");

        String email = randomEmail(20);

        studentController instance = new studentController();

        boolean expResult = true;

        boolean result = instance.checkEmail(email);

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testCheckDate

The next test is to see if the checkDate function can properly validate a date formatted String. The test generate a random String in the format of dd/mm/yyyy. If the String follows the conventional format, then it returns a boolean value “true” and the test passes.

@Test

    public void testCheckDate() {

        // check date in the dd/mm/yyyy

        System.out.println("checkDate");

        String date = randomDob();

        studentController instance = new studentController();

        boolean expResult = true;

        boolean result = instance.checkDate(date);

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testCheckPhone

The next test is to see if the checkPhone function can properly validate a phone formatted String. The test generate a random String in the format of 10 integer digits. If the String follows the conventional format, then it returns a boolean value “true” and the test passes.

@Test

    public void testCheckPhone() {

        // check phone number with a random phone number

        System.out.println("checkPhone");

        String phone = randomPhone(10);

        studentController instance = new studentController();

        boolean expResult = true;

        boolean result = instance.checkPhone(phone);

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testCheckStudent

The next test is to see if the program can properly return a student when a user search for an ID. The test read the file student.csv, make an ArrayList of Student out of it and check the id column of the first row and calls for the function checkStudent(). If a student is found, then the function returns a “true” boolean value.

@Test

    public void testCheckStudent() {

        // read the file student.csv

        // read the second column of the file as id

        // check if the id is in the student list

        // if the id is in the student list, then this test is passed

        System.out.println("checkStudent");

        studentController instance = new studentController();

        instance.getStudentList();

        String id = studentController.studentList.get(1).getId();

        boolean expResult = true;

        boolean result = instance.checkStudent(id);

        assertEquals(expResult, result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testReadStudentFromFile

The test testReadStudentFromFile() is to test the ability to read from a file and make a List of Student out of it. If the list is no longer null after reading the file, then the test passes

@Test

    public void testReadStudentFromFile() {

        // read the file student.csv

        // if student list is not null, then this test is passed

        System.out.println("readStudentFromFile");

        studentController instance = new studentController();

        instance.readStudentFromFile("student.csv", instance.studentList);

        assertNotNull(studentController.studentList);

        // TODO review the generated test code and remove the default call to fail.

    }

### testDeleteStudent

This test uses deleteStudent() to delete a student then use findStudent to check for it in the file, if the deleted student is not found anymore, the function returns false and the test passes.

@Test

    public void testDeleteStudent() {

        // read the file student.csv, get an id from the student list

        // delete the student with the id

        // if the student is deleted, then the student when finding with id is null

        // and this test is passed

        System.out.println("deleteStudent");

        studentController instance = new studentController();

        instance.getStudentList();

        String id = studentController.studentList.get(1).getId();

        instance.deleteStudent(id);

        StudentModel result = instance.findStudent(id);

        assertNull(result);

        // TODO review the generated test code and remove the default call to fail.

    }

### testDeleteAllStudent

This test uses deleteAllStudent() to delete all data on the file and then compare the file to an empty list, if they are equal, then the test passes.

@Test

    public void testDeleteAllStudent() {

        // delete all student in the student list

        // if the student list is empty, then this test is passed

        System.out.println("deleteAllStudent");

        studentController instance = new studentController();

        instance.getStudentList();

        instance.deleteAllStudent();

        // expected result = studentList

        // studentList.clear

        ArrayList<StudentModel> expResult = instance.studentList;

        expResult.clear();

        assertEquals(expResult, studentController.studentList);

        // TODO review the generated test code and remove the default call to fail.

    }

### testSortStudentByGpaAscending

This test uses sortStudentByGpaAscending() to sort the list of students in the order of GPA ascending and then goes through the list comparing each pair of student’s gpa. If the GPA of the next student is greater than the current one, the loop continues. If it keeps happening until the loop ends, it means that the list is in the correct order and the test passes.

@Test

    public void testSortStudentByGpaAscending() {

        // sort the student list by the order of gpa ascending

        // check the gpa of student from first to last, if it's in

        // ascending order, then this test is passed

        System.out.println("sortStudentByGpaAscending");

        studentController instance = new studentController();

        instance.getStudentList();

        instance.sortStudentByGpaAscending();

        ArrayList<StudentModel> result = instance.studentList;

        for (int i = 0; i < result.size() - 1; i++) {

            assertTrue(result.get(i).getGpa() <= result.get(i + 1).getGpa());

        }

        // TODO review the generated test code and remove the default call to fail.

    }

### testSortStudentByGpaDescending

This test uses sortStudentByGpaDescending() to sort the list of students in the order of GPA ascending and then goes through the list comparing each pair of student’s gpa. If the GPA of the next student is smaller than the current one, the loop continues. If it keeps happening until the loop ends, it means that the list is in the correct order and the test passes.

@Test

    public void testSortStudentByGpaDescending() {

        // sort the student list by the order of gpa descending

        // check the gpa of student from first to last, if it's in

        // descending order, then this test is passed

        System.out.println("sortStudentByGpaDescending");

        studentController instance = new studentController();

        instance.getStudentList();

        instance.sortStudentByGpaDescending();

        ArrayList<StudentModel> result = instance.studentList;

        for (int i = 0; i < result.size() - 1; i++) {

            assertTrue(result.get(i).getGpa() >= result.get(i + 1).getGpa());

        }

        // TODO review the generated test code and remove the default call to fail.

    }

# Result

Graphical user interface, application

Description automatically generated

Figure 8 Login window

Graphical user interface, application

Description automatically generated

Figure 9 Login failed

Graphical user interface, application

Description automatically generated

Figure 10 Register window

Graphical user interface

Description automatically generated

Figure 11 Register failed

Graphical user interface, application

Description automatically generated

Figure 12 Successful register

Graphical user interface

Description automatically generated

Figure 13 Successful Login

Graphical user interface, application

Description automatically generated

Figure 14 Student Index window

Graphical user interface, application

Description automatically generated

Figure 15 Search by name

Graphical user interface, application

Description automatically generated

Figure 16 Search by ID

Graphical user interface, application

Description automatically generated

Figure 17 Sort by GPA ascending

Graphical user interface, application

Description automatically generated

Figure 18 Sort by GPA descending

Graphical user interface, application

Description automatically generated

Figure 19 Student Information

Graphical user interface

Description automatically generated

Figure 20 Edit student information

A picture containing graphical user interface

Description automatically generated

Figure 21 Add a new student

Graphical user interface

Description automatically generated

Figure 22 Add Student with invalid phone number

Graphical user interface

Description automatically generated

Figure 23 Add student with invalid email format

Graphical user interface

Description automatically generated

Figure 24 Add student with invalid grade

Graphical user interface, application

Description automatically generated

Figure 25 Add Student with valid information

Graphical user interface, application

Description automatically generated

Figure 26 Student List after adding

Graphical user interface, application

Description automatically generated

Figure 27 Delete selected Student

Graphical user interface, application

Description automatically generated

Figure 28 Delete All Students

# Conclusion

# References

Tutorialspoint, 2022. *MVC Framework - Introduction.* [Online]   
Available at: https://www.tutorialspoint.com/mvc\_framework/mvc\_framework\_introduction.htm  
[Accessed 24 June 2022].