student-database-program

MACHINE PROBLEM 2

Brymer Bernard R. Meneses
Jude Michael F. Gatchalian

TABLE OF CONTENTS

INTRODUCTION

Meow Student Database is a student database which has functions to add, delete, search, and edit data entries. This database stores student information that includes their names, SAIS ID (a number identifier for the school portal), Student Number (an identification for their university information), and their address, from which a linked list is the primary data structure in storing the entries.

Java is the primary programming language that is used in this application. However, the user interface was made through a User Interface builder (Scene Builder) in which through the JavaFX libraries enabled it to be integrated to Java.

This library is a popular software platform in building desktop applications, especially for integrating rich-web applications of custom user interfaces. Moreover, Gradle is also used in this application to run it smoothly without manually configuring and installing the necessary libraries to run this application. It is an automation tool for multi-language development, even though Java is the main language used here, the user interface has integrated (Cascading Style Sheets) CSS, and FXML (a mark-up language for JavaFX applications).

This application utilizes a dashboard style of user interface that allows the user to navigate through the application's different functionalities with ease, and also aesthetics. Moreover, dialog boxes were also integrated in this application to easily communicate with the user, and how to properly utilize the application's functionality.

DESCRIPTION OF DATA STRUCTURES

The main data structure that is used in this project is the Database class (MP.Database), this data structure wraps our own implementation of the Linked List Data Structure (MP.linkedList.LinkedList)

The Database class allows for seamless saving and reading the data through a file called `database.dat`. This allows for the data to be saved and retrieved easily.

We have set this class to have a maximum length of 10 entries, each with name, SAIS ID, student number, and address; this configuration can be found in the StudentDB class itself.

When the application launches, the file databse.dat is read using the static method `Database.readSavedFile()`, this ensures that the multiple components of the code have the same data.

When a change is made to the database, for instance adding new data in the database, we call the database.writeToFile() to save changes to the database.dat file.

The bulk of the application runtime happens at the MP.StudentDB, methods such as `database.writeToFile()` listen for button presses and update the state of the application.

For a more comprehensive view of the codebase please refer to the documentation inside the code found in the different java files within the folders.

STRUCTURE OF THE CODE

Important Directories

Notes for Gradle
The project was initialized using gradle, this section briefly describes the file structure of the project.

```
├── student-database-program
│   ├── build
│   ├── gradle
│   ├── src
│   │   ├── main
│   │   │   ├── MP
│   │   │   │   ├── components
│   │   │   │   ├── interfaces
│   │   │   │   └── linkedList
│   │   │   └── resources
│   │   │       ├── fxml
│   │   │       │   └── css
│   │   │       └── icons
│   │   └── test
│   │       └── MP
│   ├── build.gradle
│   ├── gradlew
│   ├── gradlew.bat
│   └── settings.gradle
```

Important Directories

src/main

The most important part of this file structure is the src/main folder, it contains the java source codes that are used in compiling the program.

src/test

This folder contains the test functions that are used to ensure that the backend code works as expected.

src/resources

This folder contains the fxml, css and png files that are used for the frontend of the application.

gradle
This folder contains the files that are needed by gradle to work.

build
This folder contains the compiled source codes that are produced by gradle.

ALGORITHM

LinkedList Functions *Back-End*

File Location and Function Name	**Main Purpose**	**Functions**
StudentData.java_StudentData()	Creates a new node for the entry	1. Place the input to each field in the entry, from **name, saisID, studentNumber,** and **address.**
StudentDB.java_addData()	Checks the conditions of the linked list and add the input entry form the user	1. IF the entry is duplicate RETURN **false** 2. IF the entry exceeds the maximum of 10 entries RETURN **false** 3. When the entry passes the previous two filter, it will add another node to the linked list, and save the entry there **(Goes to StudentData())**
StudentDB.java_editData()	Edit the selected entry in the linked list, and save these changes in a file	1. Searches the selected entry in the **database (Linked list)** using a FOR Loop 2. When a match is found, it will initialize the entry for saving the changes. 3. A dialog box that confirms the user action will pop out. 4. When the user clicks it to save, and IF the entry is a duplicate, THEN it will clear the text fields in

		the edit panel, AND send a dialog to the user that the edited entry is a duplicate. 5. ELSE it will save the changes in the **database.dat** via **updateSavedData()** and clear the text fields in the edit panel.
Utils.java_keywordInEntries()	Checks if there are matches from the search keyword	1. Checks if the keyword matches to any **name**s in the database (Linked list), THEN RETURNS **true** 2. Checks if the keyword matches to any **address**es in the database (Linked list), THEN RETURNS **true** 3. Checks if the keyword matches to any **saisID** in the database (Linked list), THEN RETURNS **true** 4. Checks if the keyword matches to any **studentNumber** in the database (Linked list), THEN RETURNS **true**
StudentDB.java_searchData()	Loads the entries when a match is found, and shows “no matches dialog” if there’s no matched keywords	1. Removes pre-existing panes in the matched results from a recent **searchEntries** 2. Initially sets the number of search results as zero **(isSearchResultZero = true)** 3. Loops a search through the database, if there’s a match, it will initialize

		a new pane to load the entries where a match were made AND **isSearchResultZero = false;** 4. After the loop, **nothingMatchedDialog** will appear if **isSearchResultZero = true.**
StudentDB.java_populateEntries()	Updates the data entries in the tables from the Edit and Delete panels	1. Clear all panes inside the table in the edit/delete panel 2. Loop to get all the data in the database, from **name, student number, SAIS ID,** and **address.** 3. Depending on what function is calling, whether from the **Edit** or **Delete** panel, it will then show the button for each panel, either an edit or delete button, and the corresponding data for each text label between each panel. 4. Then, either a panel will be deleted from the Delete function or a panel will be updated from an Edit function.
StudentDB.java_showData()	Initializes all entries from the database, and load a panel for each entry	1. Using a loop, all entries from the database will be fetched 2. After fetching an entry, it will create a pane in a table where the function is called, and load the

		entry in that pane. These data include the n**ame, student number, SAIS ID,** and **address.** 3. The loop will repeat until it reaches the end of the number of entries within the database.
StudentDB.java_deleteData()	Delete the selected entry in the linked list, and save these changes in a file	1. Searches the selected entry in the **database (Linked list)** using a FOR Loop 2. When a match is found, it will delete the corresponding data in its entirety 3. Saves the changes in the **database.dat** via **updateSavedData().**
Database.java_writeChangesToFile()	Saves the changes made throughout the use of the application	1. Finds the file in its directories **(database.dat)** puts into a input stream 2. Opens the file 3. Save all entries in the database inside it 4. Closes the file
StudentDB.java_readSavedFile()	Reads the saved file of the database, and passes the data to a calling function	1. Finds the file in its directories **(database.dat)** puts into a input stream 2. Opens the file 3. Reads all of its contents, and save it to a object from the input stream 4. RETURNS the object, which is the contents of the database to the functions that will call it.

StudentDBDemo.java_main()	Initializes a pre-set entries in the linked list	1. Initializes the **LinkedList database** 2. Adds 5 initial entries in the database 3. IF **database.dat** does not exists, then it will save the initialized entries in the **database.dat .(writeChangesToFile)** 4. Initialize the user interface **(App.java)** 5. Load the application's user interface
App.java_initialize()_start()	Launches the program using the set parameters of it	1. Finds the "**DashFX.fxml**" 2. Hide the windows application window buttons (close, minimize, maximize) 3. Initialize the mouse input in the program 4. Set the title of the program as "**Meow Student Database**" 5. Set a new application window icon. 6. Show the application windows to the user

Dashboard Panel Pages - Button Function *Front-End*

StudentDB.java_handleClicks()

Once the user selects a button in the dashboard, it will get the source from where the click originated (**clickedButton**), and show the user the panel they selected.

Dashboard Panel Buttons

```
public void handleClicks(ActionEvent actionEvent) {
        database = Database.readSavedFile();

        Button clickedButton = (Button) actionEvent.getSource();
        String clickedButtonId = clickedButton.getId();
        Stage stage = (Stage) btnMin.getScene().getWindow();


        switch (clickedButtonId) {
            case "btnMin":
                stage.setIconified(true);
                break;
            case "btnClose":
                stage.close();
                break;
            case "btnView":
                changePage("view");
                break;
            case "btnHome":
                changePage("home");
                break;
            case "btnEdit":
                changePage("edit");
                break;
            case "btnAdd":
                changePage("add");
                break;
            case "btnSearch":
                changePage("search");
                break;
            case "btnDelete":
                changePage("delete");
                break;
            case "saveAdd":
                handleAdd();
                break;
            case "saveEdit":
                handleEdit();
                break;
        }
    }
```

Figure 1. StudentDB.java_handleClicks().

Displaying Dialog Boxes to the User - Loading a new Window Function *Front-End DialogBox.java*

The dialog boxes pop up to certain interactions within the program, this include success notifications in adding, deleting, and editing entries, as well as displaying errors like an overflow of more than 10 entries, adding a pre-existing entry, or a confirmation in the user action (edit/delete).

Notes for the FXML Documents (\src\main\resources\fxml)

This document is generated by Scene Builder, which was the development environment in designing the user interface. The table below describes the main functions used throughout the applications user interface.

Functions	**Task**
fx:id	This contains the address of a certain object in the user interface, such as the button, pane, text, and labels. This ID serves as a name to call in the dashController.java.
onAction	This serves as the function to call in the dashController.java. For example, **onAction = #handleClicks**, calls the **handleClicks()** function.

The dialog boxes pop up to certain interactions within the program, this include success notifications in adding, deleting, and editing entries, as well as displaying errors like an a Besides, below are the user interface objects used in the application.

Objects	**Purpose**

Button	This object receives the user click input in performing certain tasks, such as moving to different panels, closing a dialog box, and acts as mode of confirmation in the user action within the application.
Pane	This object is a physical container that groups objects, like a collection of Texts and a button.
VBox	This object is a vertical column that stacks the pane neatly on top of each other.
Text	This object displays text, either as a pre-set label, or a container of the entries in the database.
ScrollPane	As a navigator for seeing all the entries within a collection of panes.

Notes for the JavaFX functions

This table shows the JavaFX functions used in the **DashController.java**, which connects the user-interface, and the backend of the application.

Functions	**Task**
setVisible	When set to **true,** it will show the object (panes, buttons, texts) visibly in the user, together with its functions. Conversely, when it is set to **false**, it will hide the corresponding object, and disable its functions.
getText	This fetches the text currently displayed to the user. This function can be applied on text fields, and texts.
getChildren	This method is used to get the children components(such as checkboxes, buttons etc) in

	a container.
setStyle	This sets a CSS in an object, in this case, it sets the highlight of the text field as shown in this code: **setStyle("-fx-border-color: #6a7281;-fx-border-radius: 5; -fx-background-color: #1a1d20;");**

ERROR HANDLING

In the *Add an Entry*, and *Edit an Entry* function…

The save button (**saveAdd** & **saveEdit** buttons) contains several error handling algorithms that sift the user's input for an initial database entry.

First, it checks whether the the fields for a **name** input are valid **roman numeral characters**, and spaces; and if not, it sends a prompt in the "Add an Entry Pane" to only input characters (**"charOnly1" displays "Characters only above the text field for name"**).

Character Filter Check

```
public static boolean isCharOnly(String input) {
        return input.matches("[a-zA-Z\\s]+");
    }
```

Figure 2. Utils.java_isCharOnly() Function.

Besides, for the case of the text field for the **SAIS ID** and **student number** only allows **numbers with no spaces** as a valid input, and if not, it also sends a prompt to only input numbers and no spaces in between the user's input **("numOnly1, numOnly2" displays "Numbers only" above the text fields for SAIS ID, and student number)**.

Numbers Filter Check

```java
public static boolean isNumberOnly(String input) {
        return input.matches("[0-9]+");
    }
```

Figure 3. Utils.java_isNumberOnly() Function.

In addition, if the user did not add input to one of the text fields, which are receiving the name, SAIS ID, student number, and address, it will prompt the user that all those fields are required **(requireNotif)**. Together with setting **requireNotif** visible to the user, which is a prompt message saying "Please complete all required fields", and highlights an empty textfield.

Empty Textfield Filter

```java
if (inputStudentNumber.isBlank()) {
            Utils.setStyleWarning(studentNumberTF);
        } else {
            Utils.setStyleNormal(studentNumberTF);
        }

        if (inputName.isBlank()) {
            Utils.setStyleWarning(nameTF);
        } else {
            Utils.setStyleNormal(nameTF);
        }

        if (inputSaisId.isBlank()) {
            Utils.setStyleWarning(saisIdTF);
        } else {
            Utils.setStyleNormal(saisIdTF);
        }

        if (inputAddress.isBlank()) {
            Utils.setStyleWarning(addressTF);
        } else {
            Utils.setStyleNormal(addressTF);
        }
```

Figure 4. Utils.java_validateInputs() Function.

Highlight Textbox

```
public static void setStyleWarning(TextField input) {
        input.setStyle("-fx-border-color: #ff6767;-fx-border-radius: 5; -fx-
background-color: #1a1d20;");
    }
```

Figure 5. Utils.java_setStyleWarning() Function.

Also, if the user inputs an out-of-range number which are numbers exceeding the range of an Integer value, it will notify the user to re-input a number within its range.

Integer Input Range Checker

```
public static boolean isOutOfRange(String saisId, String studentNumber) {
        try {
            Integer.parseInt(saisId);
            Integer.parseInt(studentNumber);
            return false;
        } catch (NumberFormatException e) {
            return true;
        }
    }
```

Figure 6. Utils.java_isOutOfRange() Function.

Furthermore, if the initial entry is a pre-existing entry, it will prompt the user that it is already in the database, and will no longer save the new entry **(notif_added)**. Also, when the database already has 10 entries, it will prompt the user that it cannot save its initial entry because the database is already full, and prompts the option to the user to delete some entries to give space to new ones **(warn_overflow)**. Note the same implementation in the **handleEdit().**

Entry Saving Filter

```
if (Utils.isDuplicate(database,student)) { // for checking for duplicate entries
                dialogBox.setConfirmButtonAction(clearTextFields);
                dialogBox.load("warn_duplicate_for_add");
                return;
            }

            if (database.length + 1 > maxStorageLength) { // For checking the
capacity of the database
                dialogBox.setConfirmButtonAction(clearTextFields);
                dialogBox.load("warn_overflow");
            } else {  //For when the entries has an available slot in the database
and a unique entry

                dialogBox.setConfirmButtonAction(()-> {
                    clearTextFields.call();
                    addData(student);
                });

                dialogBox.load("notif_add_success");
            }
```

Figure 7. StudentDB.java_handleAdd() Function.

Therefore, when the user both sends a valid input to each respective text field, has an allocated space for it, and it's a unique entry, then it will properly save it to the student database, and prompt the user that it has successfully added that initial entry **(notif_addsuccess).**

In the *Search an Entry* function…

The Search bar can detect if the user wishes to search an entry by pressing Enter (*this function is embedded in the* ***searchField*** *in* ***DashFX.fxml*** *pointing as well in* ***handleSearch()***) or clicking the Search button icon **(keyPress())**.

Searching Function

```
public void searchData(String toSearch) {
        searchEntries.getChildren().clear();//removes pre-existing results if
there's any

        boolean isSearchResultZero = true;
        for (int i=0; i<database.length; i++) {
            StudentData element = database.get(i); //search for matches in the
database

            if (Utils.keywordInEntries(database.get(i), toSearch)) {
                DatabaseEntry entry = new DatabaseEntry("search", element);
                searchEntries.getChildren().add(entry);// if there's a match found
in the database, creates a new pane where a new match is found
                isSearchResultZero = false;
            }

        }

        nothingMatchedDialog.setVisible(isSearchResultZero);//displays the "No
match" dialog whether a match is found
    }
```

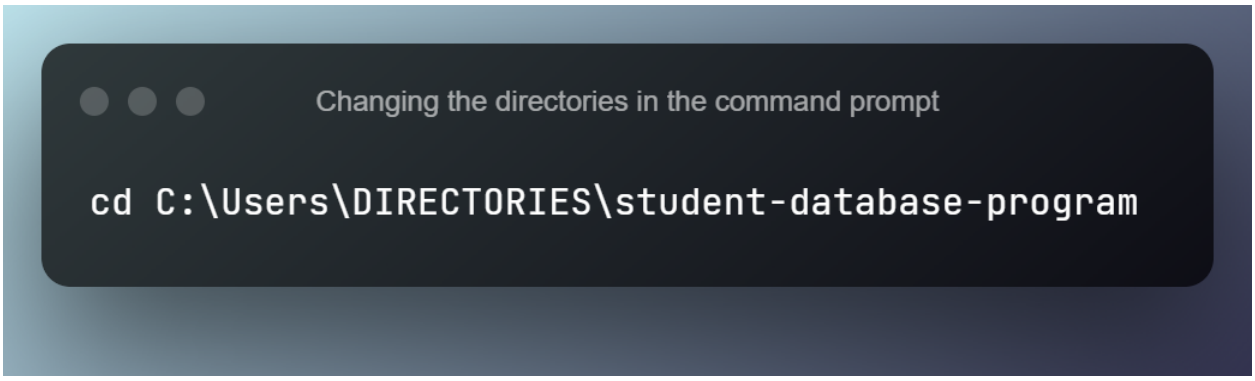
Figure 8. StudentDB.java_searchData() Function.

When these actions are done, it catches the user's input in the textfield. If it's completely blank, it will show all the searchable entries in the database. But if there's a character or number input, the algorithm will sift through all the entries **(searchData())**, to each of its components, and if there's a match, it will show all the results. If there's no match from the user input, it will prompt the user that no matches were found in the database **(nothigMatchedDialog)**.

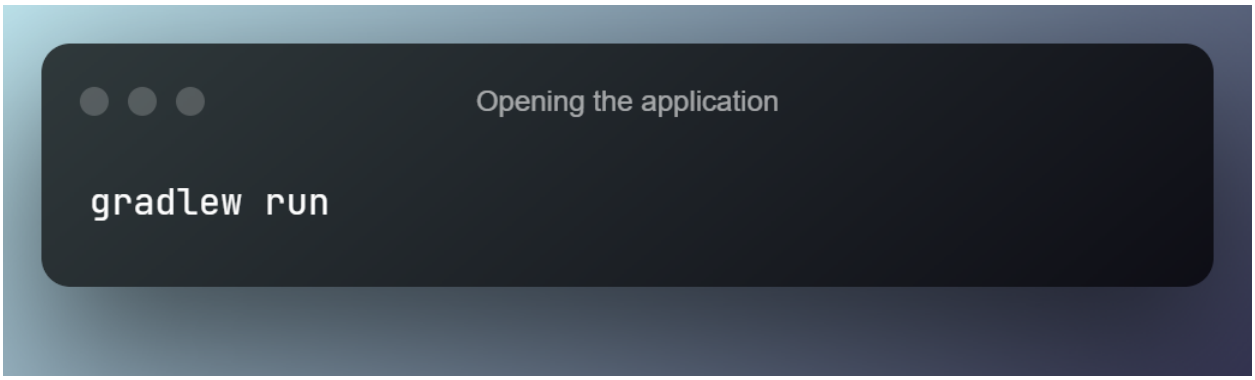
APPLICATION MANUAL

How to run the application?

1. Open a command prompt, and change its directory to where the folder of "***student-database-program***" is located.



2. Type "***gradlew run***", and then press Enter to initialize, compile, and run the application. This command also facilitates the installation and configuration of JavaFX.



3. Voila, the application is up and running.