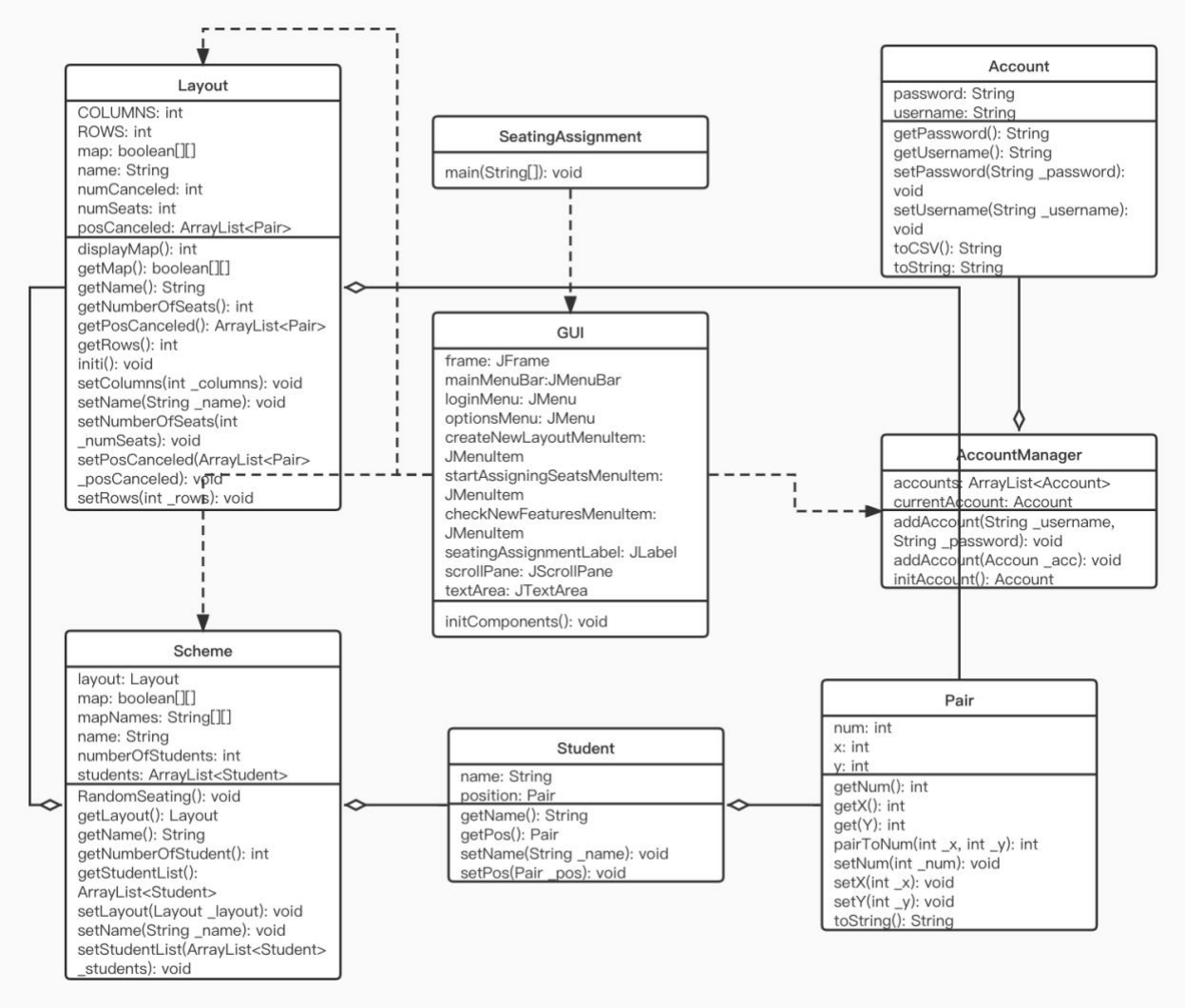
CRITERION C: DEVELOPMENT

UML Diagram

I will code the program using object-oriented programming, consist of 8 classes and each implements the use of encapsulation and polymorphism. This UML diagram shows the class components (instance variables and methods) and relationships between classes.



List of Techniques

The techniques used in the program include:

- Arrays
- Try & catch exception handling
- File handling
- Graphical user interface

- Complex selection with nested if or multiple conditions
- Global and local variables
- Sequential search algorithm
- Encapsulation, using get and set methods

Arrays

A lot of arrays are used throughout my code. Array is a very efficient type of data type that stores a row of data with the same identifier name and type. For example, in Scheme class, I use the two-dimensional array map and mapNames to store the seating plan. I use the two-dimensional array because usually the desks in a classroom are positioned in a rectangular shape that has two attributes: row and column. The first index corresponds to the row number, valid in range [1,ROWS], and the second index corresponds to the column number, valid in range [1,COLUMNS], in which ROWS is the row number and COLUMNS is the column number of the layout. If either index is 0, then the program would have logic error because that variable is meaningless. If either index is greater than ROWS (or COLUMNS), then an ArrayOutOfBoundException will be thrown, causing the program to crash. It is much more efficient to manage the whole seating plan using the two arrays because the state of each position can be visited using the row and column number.

```java
public class Scheme {

    private String name;
    private Layout layout;
    private ArrayList<Student> students;
    private int numberOfStudents;
    private boolean[][] map;
    private String[][] mapNames;

    /**
     * Default constructor
     * Create a default scheme
     */
    public Scheme() {
        name="";
        layout=null;
        students=new ArrayList<Student>();
        numberOfStudents=-1;
        map=new boolean[0][0];
        mapNames=new String[0][0];
    }
```

Try & Catch Exception Handling

To prevent crash of the program, exception is thrown when there is an invalid process. The program will crash to indicate possible causes for the exception. This helps me debug and improve the program in the future.

In RandomSeating method in Scheme class, the program catches FileNotFoundException when namelist.csv does not exist. Message is outputted to the screen to inform the client that he or she hasn't created the namelist yet.

```
public void RandomSeating() throws Exception {
    //initialize
    String username=AccountManager.currentAccount.getUsername();
    String thePath="Accounts/"+username+"/namelist.csv";
    Scanner fileReader = null;
    Scanner scn = new Scanner(System.in);
    int ok=1;

    //check if namelist.csv exists
    try {
        fileReader = new Scanner(new File(thePath));
    } catch(FileNotFoundException e) {
        ok=0;
        System.out.println("Input the namelist at "+thePath);
    }
    if(ok==0) {
        System.out.println("Please input your namelist now, otherwise the program may crash! Input anything to continue ass
        File file1 = new File(thePath);
        file1.createNewFile();
        String emptyStr=scn.next();
    }

    //read in the student namelist
    fileReader = new Scanner(new File(thePath));
    while (fileReader.hasNext()) {
        String aName=fileReader.nextLine();
        //System.out.println("The student name is "+aName);
```

Also, when the program output to a file, try-catch clause is used in case there is an IOException. Exception message will be recorded in a log. This works the same for the initi method in Layout class, and initAccount method in AccountManager class.

```
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(new File("Accounts/"+username+"/seating.csv")));
    for(int i=1;i<=layout.ROWS;i++) {
        for(int j=1;j<=layout.COLUMNS;j++) {
            writer.write(mapNames[i][j] + ",");
        }
        writer.write("\n");
        writer.flush();
    }
    writer.close();
} catch (IOException ex) {
    Logger.getLogger(SeatingAssignment.class.getName()).log(Level.SEVERE, null, ex);
}
```

```
    System.out.println("Are you satisfied with this layout? Press 1 if you like and the layout will be saved; press 0 to c
    int opt=scn.nextInt();
    if(opt==1) {
        try {
            //write the configuration into a txt file
            String pathName = "Accounts/" + AccountManager.currentAccount.getUsername() + "/" + myLayout.name + ".txt";
            File file1 = new File(pathName);
            file1.createNewFile();
            BufferedWriter writer = new BufferedWriter(new FileWriter(pathName));
            //writer.write(myLayout.name + "\n");
            writer.write(myLayout.ROWS + "\n");
            writer.write(myLayout.COLUMNS + "\n");
            writer.write(myLayout.numCanceled + "\n");
            for(Pair pr:myLayout.posCanceled)
                writer.write(pr+"\n");
            writer.close();
        } catch (IOException ex) {
            Logger.getLogger(Layout.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println("Your layout named "+myLayout.name+" is successfully saved.");
    }
    else if(opt==0) {
        //restart the initialization process
        initi();
    }
    else {
        throw new Exception("Invalid option!");
    }
}
```

```
    try {
        //add the new account to Accounts.csv
        BufferedWriter csvWriter = new BufferedWriter(new FileWriter(new File("Accounts.csv")));
        for(Account _acc:accounts) {
            csvWriter.write(_acc.toCSV() + "\n");
            csvWriter.flush();
        }
        csvWriter.close();
        //add a new directory containing the Accounts information
        File f=new File("Accounts/"+myUsername);
        f.mkdir();

    } catch (IOException ex) {
        Logger.getLogger(AccountManager.class.getName()).log(Level.SEVERE, null, ex);
    }
    currentAccount=myAcc;


    try {
        //input student namelist.csv
        BufferedWriter csvWriter = new BufferedWriter(new FileWriter(new File("Accounts/"+myUsername+"/namelist.csv
        csvWriter.write("<Input student namelist here. One student per line.>\n");
        csvWriter.flush();
        csvWriter.close();
    } catch (IOException ex) {
        Logger.getLogger(AccountManager.class.getName()).log(Level.SEVERE, null, ex);
    }
    System.out.println("Import student list at "+"Accounts/"+myUsername+".csv");
```

In main method in SeatingAssignment class, FileNotFoundException and Exception are used because the methods that this method calls may throw those exceptions.

```
public static void main(String[] args) throws FileNotFoundException, Exception {

    Scanner scn=new Scanner(System.in);
    System.out.println("Welcome to the Seating Assignment Program, develoepd by Stanley!");

    Account myAcc=AccountManager.initAccount();
    if(myAcc==null) throw new Exception("Please login again.");
    System.out.println("Hi, "+AccountManager.currentAccount.getUsername());

    System.out.println("Options:\n1. create a new layout\n2. start assigning seats\n3. check features");

    int opt=scn.nextInt();
    if(opt==1) {
        Layout newLayout=new Layout();
        //a new layout is added to the account
    }
    else if(opt==2) {
        Scheme newScheme = new Scheme();
        newScheme.RandomSeating();//start assigning seats
    }
    else if(opt==3) {
        System.out.println("Current feature: random seating.\nOther features not implemented.");
    }
    else {
        throw new Exception("Invalid option!");
    }
}
```

File Handling

I choose to use file input and output due to several advantages. First, reading from and writing to files in local devices is fast and convenient. It does not require the internet connection, and the client can directly edit them. Second, file input and output functions work for all platforms as the files used in my program (.txt and .csv) are well-supported across platforms. Third, my program shows the separation of front-end and back-end, because the program deals with computation of data separate from the input-and-output process.

In initAccount method in AccountManager class, the program first reads all accounts stored in Accounts.csv. Then, the client input username and password. If the client selects “create a new account”, then the program writes the new account in Accounts.csv, and creates a directory with path Accounts/<username>/. If the client selects “delete an account”, then the program finds the account and delete it from Accounts.csv, and the account’s directory is deleted from the computer.

```
     * @throws Exception
     */
    public static Account initAccount() throws FileNotFoundException, Exception {
        accounts=new ArrayList<Account>();

        //load every previous accounts
        Scanner terminalScanner=new Scanner(System.in);

        //file input
        Scanner csvScanner = new Scanner(new File("Accounts.csv"));
        while(csvScanner.hasNext()) {
            String[] nextUserInfo=csvScanner.nextLine().split(SEPARATOR);
            AccountManager.addAccount(nextUserInfo[0], nextUserInfo[1]);
        }
        csvScanner.close();

        int opt;
        String myUsername,myPassword;
        System.out.println("Input 1 to create a new account; input 2 to log in to the system; input 3 to delete an account."
        opt=terminalScanner.nextInt();
        if(opt==1) {//create a new account

            while(true) {
                System.out.print("Create a new account:\nInput username: ");
                myUsername=terminalScanner.next();
                System.out.print("Input password: ");
                myPassword=terminalScanner.next();
```

Create a new account:

```
        try {
            //add the new account to Accounts.csv
            BufferedWriter csvWriter = new BufferedWriter(new FileWriter(new File("Accounts.csv")));
            for(Account _acc:accounts) {
                csvWriter.write(_acc.toCSV() + "\n");
                csvWriter.flush();
            }
            csvWriter.close();
            //add a new directory containing the Accounts information
            File f=new File("Accounts/"+myUsername);
            f.mkdir();

        } catch (IOException ex) {
            Logger.getLogger(AccountManager.class.getName()).log(Level.SEVERE, null, ex);
        }
        currentAccount=myAcc;


        try {
            //input student namelist.csv
            BufferedWriter csvWriter = new BufferedWriter(new FileWriter(new File("Accounts/"+myUsername+"/namelist.csv")
            csvWriter.write("<Input student namelist here. One student per line.>\n");
            csvWriter.flush();
            csvWriter.close();
        } catch (IOException ex) {
            Logger.getLogger(AccountManager.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println("Import student list at "+"Accounts/"+myUsername+".csv");
```

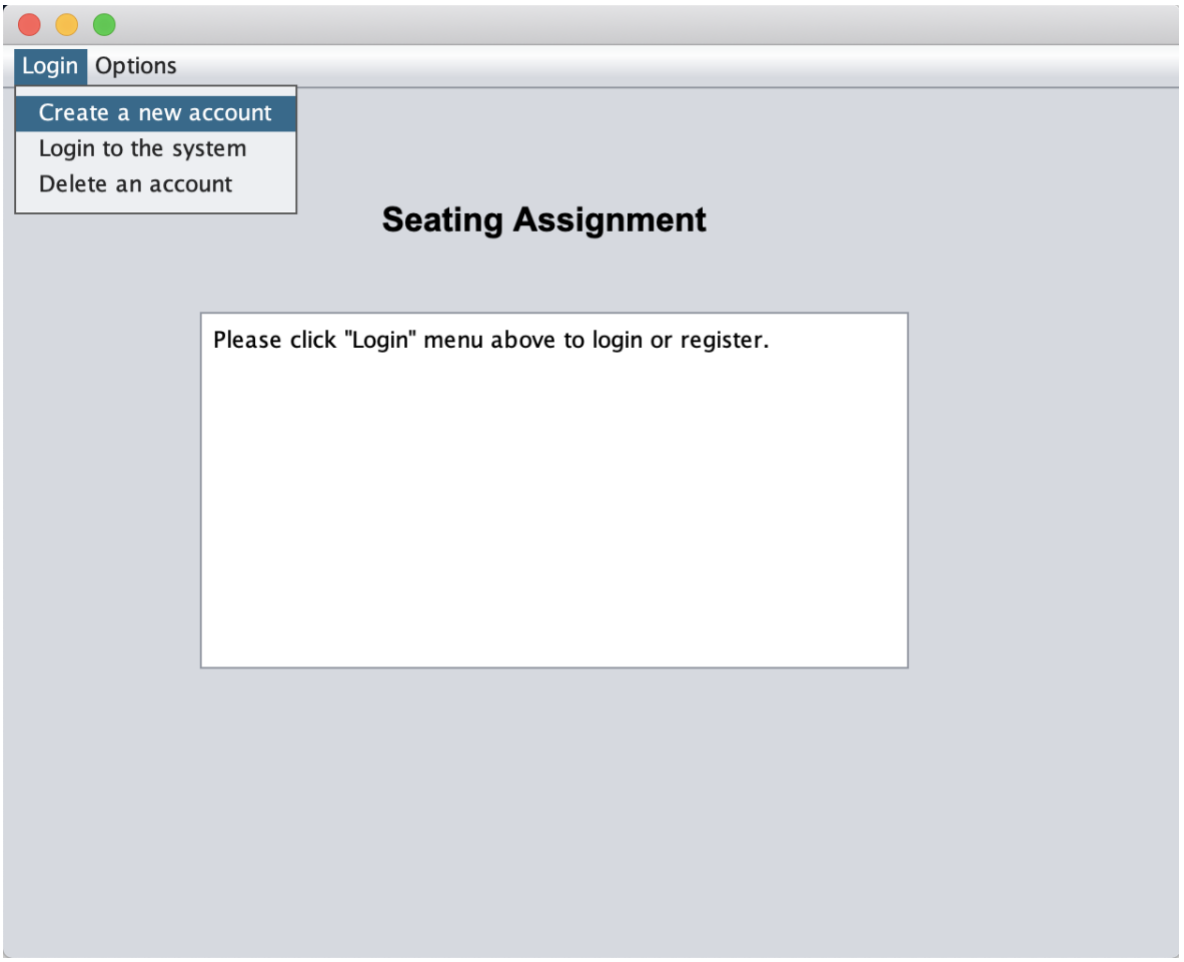
Delete an account:

```
            System.out.println("Account "+accounts.get(i).getUsername()+" is canceled");
            accounts.remove(i);

            try {
                BufferedWriter csvWriter = new BufferedWriter(new FileWriter(new File("Accounts.csv")));
                for(Account _acc:accounts) {
                    csvWriter.write(_acc.toCSV() + "\n");
                    csvWriter.flush();
                }
                csvWriter.close();
            } catch (IOException ex) {
                Logger.getLogger(Account.class.getName()).log(Level.SEVERE, null, ex);
            }

            String pathName="Accounts/"+_userName+"/";
            System.out.println("The path is: "+pathName);

            File folder = new File(pathName);
            File[] userFiles = folder.listFiles();
            for(File file : userFiles) {
                file.delete();
            }

            folder.delete();
            System.out.println(folder.exists());

            ok=1;
            break;
```

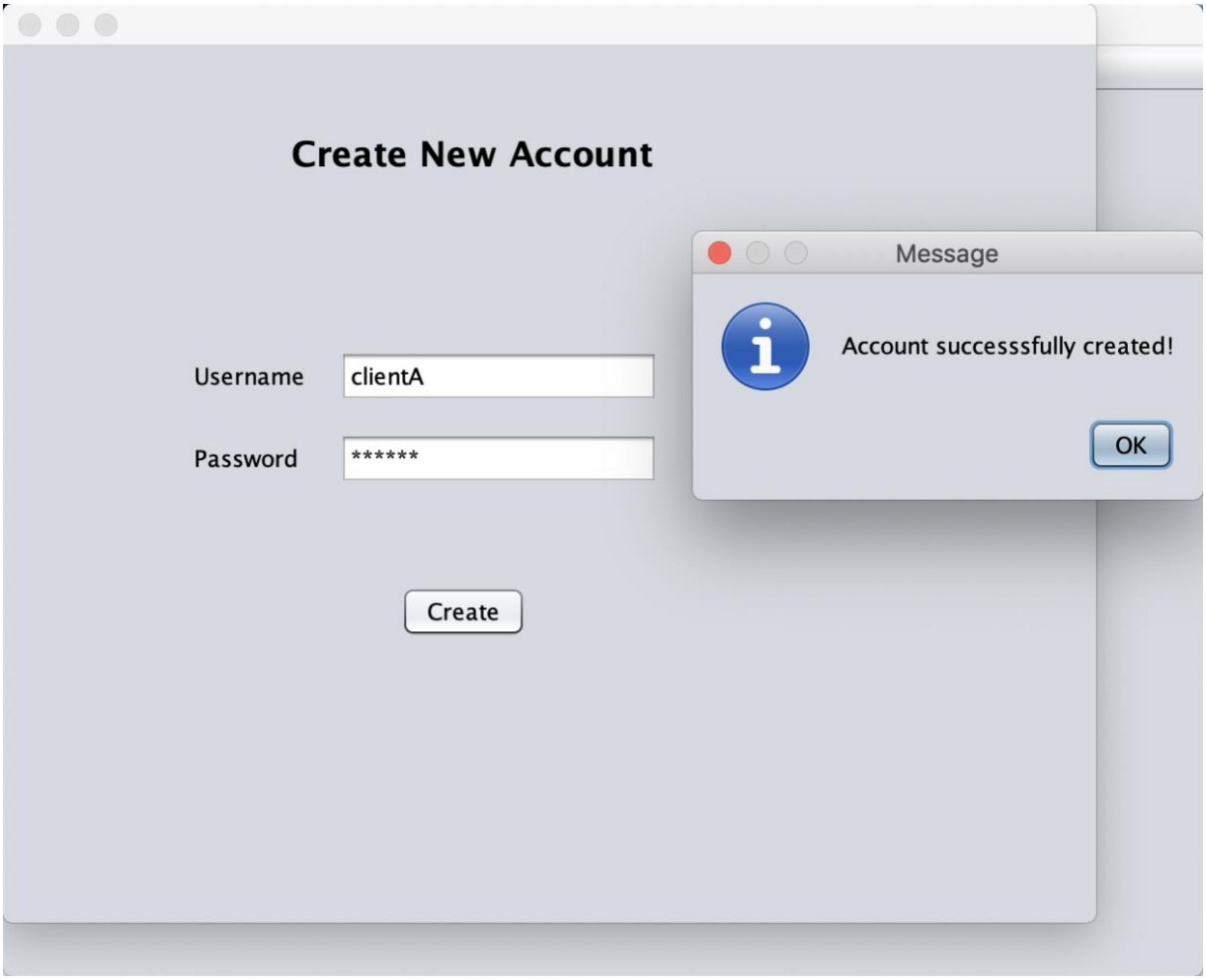
Graphical User Interface

The program uses Graphical User Interface (GUI) because it is much more approachable and elegant compared with command-line user interface. The client can click buttons and access different menus when using the product.

When first seeing the product, the user must click “Login” menu to login or register.



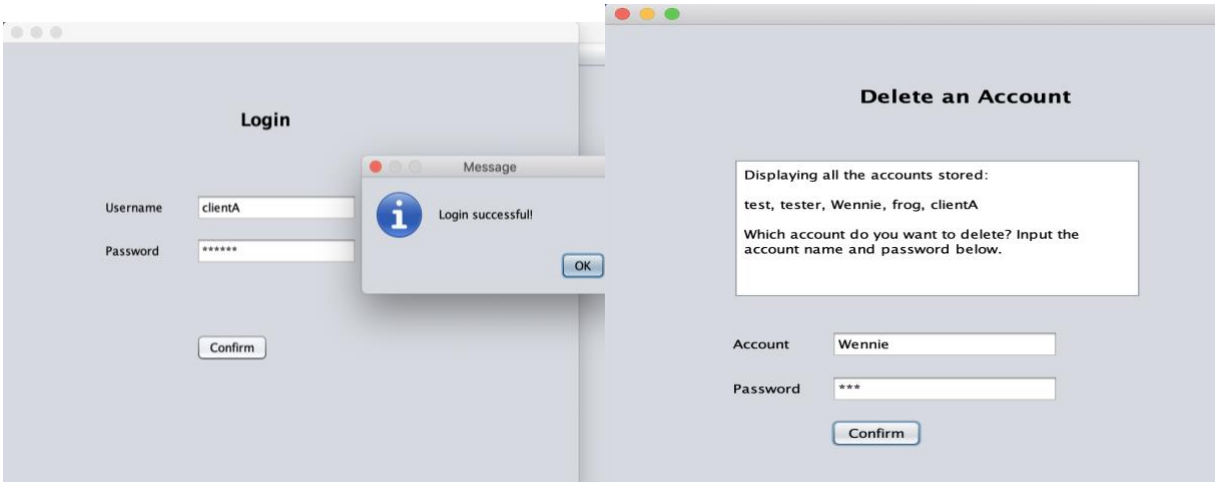
If the client chooses “create a new account”, then username and password for the account is required. If the client clicks “Create” button, the program checks if the inputted username matches with any account in Accounts.csv. If no, then a message saying “Account successfully created” will pop up.



When the user selects “create a new account”, createNewAccountMenuItemActionPerformed method in GUI class is called. The method will set createNewAccountFrame as visible, then load all accounts stored in Accounts.csv. The client may input data in createNewAccountFrame.

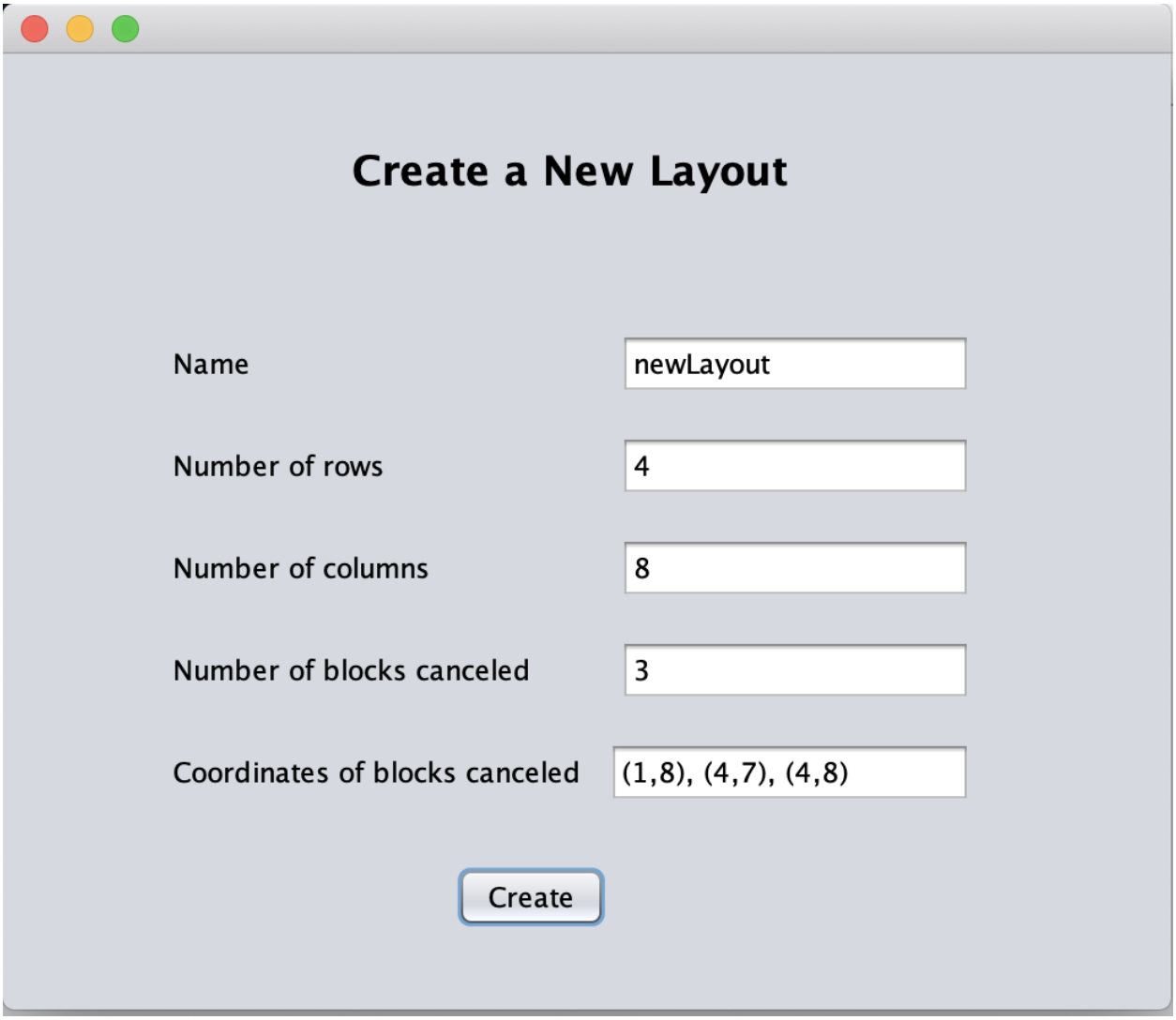
```
private void createNewAccountMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    clearTextComponents(createNewAccountFrame);
    createNewAccountFrame.setVisible(true);
    createNewAccountFrame.pack();

    accounts=new ArrayList<Account>();
    //Scanner terminalScanner=new Scanner(System.in);
    Scanner csvScanner = null;
    try {
        csvScanner = new Scanner(new File("Accounts.csv"));
    } catch (FileNotFoundException ex) {
        Logger.getLogger(GUI.class.getName()).log(Level.SEVERE, null, ex);
    }
    while(csvScanner.hasNext()) {
        String[] nextUserInfo=csvScanner.nextLine().split(SEPARATOR);
        AccountManager.addAccount(nextUserInfo[0], nextUserInfo[1]);
    }
    csvScanner.close();


}
```

The user can also select “Login” or “Delete an account”. The first option has almost the same functionality as the “create new account” option, while the second option would display the accounts stored.

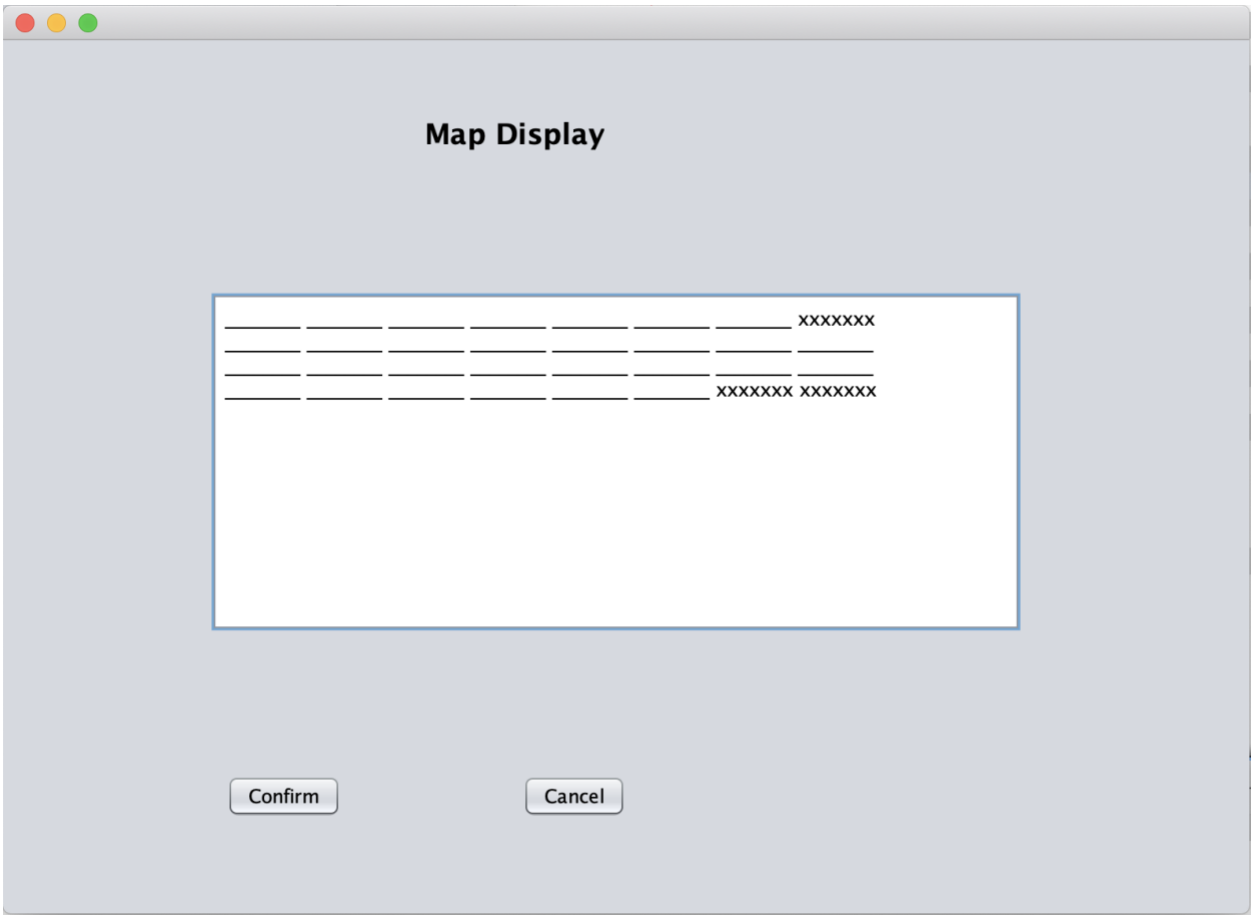


After the client login to the system, a new layout can be created. createNewLayoutMenuItemActionPerformed method in GUI class will be called, and createNewLayoutFrame is set visible and the client can input data for the new layout.

```
private void createNewLayoutMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    clearTextComponents(createNewLayoutFrame);
    createNewLayoutFrame.setVisible(true);
    createNewLayoutFrame.pack();
}
```



After the client input data for the new layout, the program displays the map, and the client can choose whether to conform and save the layout or to cancel and create the layout again. If the client selects “save”, then a .txt file with the name assigned by the user will be created with those input.



Complex Selection with Nested If or Multiple Conditions

Selections in my program could be nested if there are options under an option. For example, in initAccount method in AccountManager class, the first if statement checks if the user want to login to the system. Then, another if statement in a while loop is used to check if the username is present in the database.

```
}
else if(opt==2) {//login
    if(accounts.isEmpty()) {
        throw new Exception("No registered users! How can you possibly login!");
    }
    while(true) {
        System.out.println("Please log in to the system: ");
        System.out.print("Input username: ");
        myUsername=terminalScanner.next();
        System.out.print("Input password: ");
        myPassword=terminalScanner.next();

        for(Account acc:accounts) {
            if(acc.getUsername().equals(myUsername) && acc.getPassword().equals(myPassword)) {
                System.out.println("Login Successful!");
                currentAccount=acc;
                return acc;
            }
        }
        System.out.println("Username and password don't match. Input again.");
    }
}
else if(opt==3) {//delete an existing account
```

When the client selects to login to the system, an if statement with multiple conditions is used. This checks that the user inputted username and password both match with data in the database.

```
}
else if(opt==2) {//login
    if(accounts.isEmpty()) {
        throw new Exception("No registered users! How can you possibly login!");
    }
    while(true) {
        System.out.println("Please log in to the system: ");
        System.out.print("Input username: ");
        myUsername=terminalScanner.next();
        System.out.print("Input password: ");
        myPassword=terminalScanner.next();

        for(Account acc:accounts) {
            if(acc.getUsername().equals(myUsername) && acc.getPassword().equals(myPassword)) {
                System.out.println("Login Successful!");
                currentAccount=acc;
                return acc;
            }
        }
        System.out.println("Username and password don't match. Input again.");
    }
}
else if(opt==3) {//delete an existing account
```

Global and Local Variables

Global variables are accessible by all classes and methods. An example would be the currentAccount variable in AccountManager class. After login to the system, future seating assignment process need to know which account the client is using, so future classes need to access the currentAccount variable.

```
public class AccountManager {
    public static final String SEPARATOR = ",";
    public static ArrayList<Account> accounts = new ArrayList<Account>();
    public static Account currentAccount;
```

In “creating a new account” section under initAccount method in AccountManager class, the client creates a new local Account variable called myAcc using inputted username and password. Having this local variable is because it will be returned to the main method in SeatingAssignment class. myAcc won’t be used in other classes or methods, and the only object used is a copy of myAcc, or the global variable currentAccount. Therefore, myAcc can be abandoned once the return statement is executed, so it is declared as a local variable.

```
System.out.print("Create a new account:\nInput username: ");
myUsername=terminalScanner.next();
System.out.print("Input password: ");
myPassword=terminalScanner.next();

int ok=1;
for(Account acc:accounts) {
    if(acc.getUsername().equals(myUsername)) {
        System.out.println("Account already exist! Create your account again.");
        ok=0;
        break;
    }
}
if(ok==0) continue;
System.out.println("Account successsfully created!");
Account myAcc=new Account(myUsername,myPassword);
AccountManager.addAccount(myAcc);
```

The use of global and local variables allowed me to isolate the global variables that were needed at more than one time in calculations and methods in multiple classes.

Sequential Search Algorithm

My program may search if one object exists in a collection of objects. Sequential search algorithm used here is to search every object in the collection in a certain order and check if the external object equals with the object in the collection. An example would be the “create a new account” part in AccountManager class. I use for-each loop to enumerate acc in accounts, the latter one being the ArrayList of type Account. Then check if acc’s username equals the username inputted by the client using getUsername method and equals method provided by String class.

```
if(opt==1) {//create a new account

    while(true) {
        System.out.print("Create a new account:\nInput username: ");
        myUsername=terminalScanner.next();
        System.out.print("Input password: ");
        myPassword=terminalScanner.next();

        int ok=1;
        for(Account acc:accounts) {
            if(acc.getUsername().equals(myUsername)) {
                System.out.println("Account already exist! Create your account again.");
                ok=0;
                break;
            }
        }
        if(ok==0) continue;
        System.out.println("Account successsfully created!");
        Account myAcc=new Account(myUsername,myPassword);
        AccountManager.addAccount(myAcc);
```

Encapsulation, Using Get and Set Methods

The idea of encapsulation is to use public get and set methods to access or modify private variables so that changes can be made outside the class but at the same time access is restricted and clearly defined by those public methods. Encapsulation promotes maintenance because code changes can be made independently without affecting other classes. Encapsulation is implemented throughout multiple classes in my program.

For example, in Account class:

```java
/**
 * Accessor
 * Get the value of the username
 * @return username
 */
public String getUsername() {
    return username;
}

/**
 * Mutator
 * Set the value of the username to _username
 * @param _username username value assigned
 */
public void setUsername(String _username) {
    username=_username;
}

/**
 * Accessor
 * Get the value of the password
 * @return password
 */
public String getPassword() {
    return password;
}

/**
 * Mutator
 * Set the value of the password to _password
 * @param _password password value assigned
 */
public void setPassword(String _password) {
    password=_password;
}
```

In Layout class:

```java
/**
 * Accesssor
 * Get the name of the layout
 * @return the name of the layout
 */
public String getName() {
    return name;
}

/**
 * Mutator
 * Set the name to _name
 * @param _name name value assigned
 */
public void setName(String _name) {
    name=_name;
}

/**
 * Accessor
 * Get the row number of the layout
 * @return row number of the layout
 */
public int getRows() {
    return ROWS;
}

/**
 * Mutator
 * Set the row number of the layout to _rows
 * @param _rows row number assigned
 */
public void setRows(int _rows) {
    ROWS=_rows;
}

/**
 * Accessor
 * Get the column number of the layout
 * @return column number of the layout
 */
public int getColumns() {
    return COLUMNS;
}
```

In Pair class:

```
/**
 * Mutator
 * Set the x coordinate of the pair to _x
 * @param _x x coordinate assigned
 */
public void setX(int _x) {
    x=_x;
}

/**
 * Accessor
 * Get the y coordinate of the pair
 * @return y coordinate of the pair
 */
public int getY() {
    return y;
}

/**
 * Mutator
 * Set the y coordinate of the pair to _y
 * @param _y y coordinate assigned
 */
public void setY(int _y) {
    y=_y;
}

/**
 * Accessor
 * Get the single number coordinate of the pair
 * @return single number coordinate of the pair
 */
public int getNum() {
    return num;
}

/**
 * Mutator
 * Set the single number coordinate of the pair to _num
 * @param _num single number coordinate assigned
 */
public void setNum(int _num) {
    num=_num;
}
```

These methods allow clients to retrieve and modify attributes of an object from any outside classes despite the fact that all instance variables in the Item class is set private. It shows the idea of abstract thinking in OOP.

(1319 words)