

OOP (CS F213) PROJECT- 1

Laundromat Management System

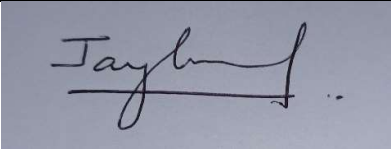
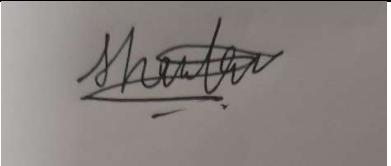
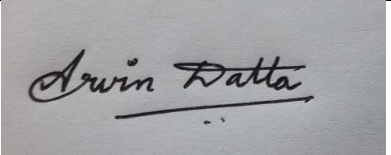
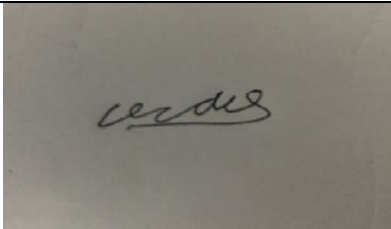
Group 21:

NAME	ID
Jay Goyal	2021A7PS2418P
Shantanu Ambekar	2021A7PS2540P
Arvin Datta	2021A7PS0430P
Vidit Shekhar Benjwal	2021A7PS0004P

Anti – Plagiarism Statement

We have read and understood the rules on plagiarism. We hereby declare that this project is the result of our own independent work, and no piece of code has been copied or utilised in any form from sources available online or elsewhere. The code in the project has not been written for any previous project.

Contribution of Group Members

Name	Contribution	Signature
Jay Goyal	Initial draft, Multithreading, Admin functionalities, and implementations of basic functionalities	
Shantanu Ambekar	File IO, Student GUI, Student functionalities, implementations of basic functionalities	
Arvin Datta	Documentation, Analysing with SOLID Principles, Identifying a design pattern	
Vidit Shekhar Benjwal	Admin GUI class and functionalities	

Use of Design Pattern

The design pattern that has been used is the Bridge Pattern, a type of structural design pattern.

The Student class and the Admin class interacts with the StudentGUI and the AdminGUI classes respectively. These classes act as an intermediary between them and the other classes like DropWashGUI and ReceiveWashGUI. This helps in decoupling the abstraction from the implementation of the functionalities in the GUI classes. In the following pictures, the abstraction is shown in ReceiveWashGUI and DropWashGUI.

```
public void receiveWash(String date) {
    for (Wash wash : plan.getWashList()) {
        if (date.equals(wash.getDateGiven())) {
            if (wash.getStatus().equals("On Delivery")) {
                wash.setStatus("Delivered");
                Swing_classes.show_message("The laundry has been received !!!");
                synchronized (studentFileWriter.writeLock) {
                    try {
                        studentFileWriter.writeStudentToFile(student: this, callIfAlreadyExists: true);
                    } catch (IOException e) {
                        System.out.println(e.getMessage());
                    } finally {
                        studentFileWriter.writeLock.notify();
                    }
                }
                return;
            } else {
                Swing_classes.show_message("Laundry status: " + wash.getStatus() + ". Wait till status is marked On Delivery");
                return;
            }
        }
    }
    Swing_classes.show_message("You did not drop a wash on this date!");
}
```

```
public void communicateDropData(String ID, double weight, String today) {
    synchronized (studentFileWriter.writeLock) {
        student = studentFileWriter.readStudentFromFile(ID);
        if (student == null) {
            Swing_classes.show_message("Student does not exist");
            return;
        }
        Student.studentFileWriter = studentFileWriter;
        studentFileWriter.writeLock.notify();
    }
    dropData = new DropData(weight, today);
    shouldRun = true;
    t.start();
}
```

The StudentGUI and the AdminGUI handle the multithreading also, allowing to hide the implementation. The client remains unaffected if there is a modification in the implementation.

Analysis of code with SOLID Principles

1. Single Responsibility:

There are student GUI and admin GUI class controls the form of input, while the Student and Admin classes, each have their own functionalities. If any function of either Student class or Admin class has to change, only that class has to be modified without any change in other classes.

2. Open-Closed Principle:

In our code, there is a User class, which is a public abstract class, containing protected instance variables. The protected keyword before the instance variables increases extensibility. The User class is then extended by both Student and Admin classes. Modifications are not done in User class, rather they are done in the extended classes based on the required functionalities.

3. Liskov's Substitution Principle:

When the User class was extended to Student and Admin classes, the functionalities in the form of methods and instance variables remain unchanged. So, a property applicable to a User instance , is also applicable to a Student or Admin instance.

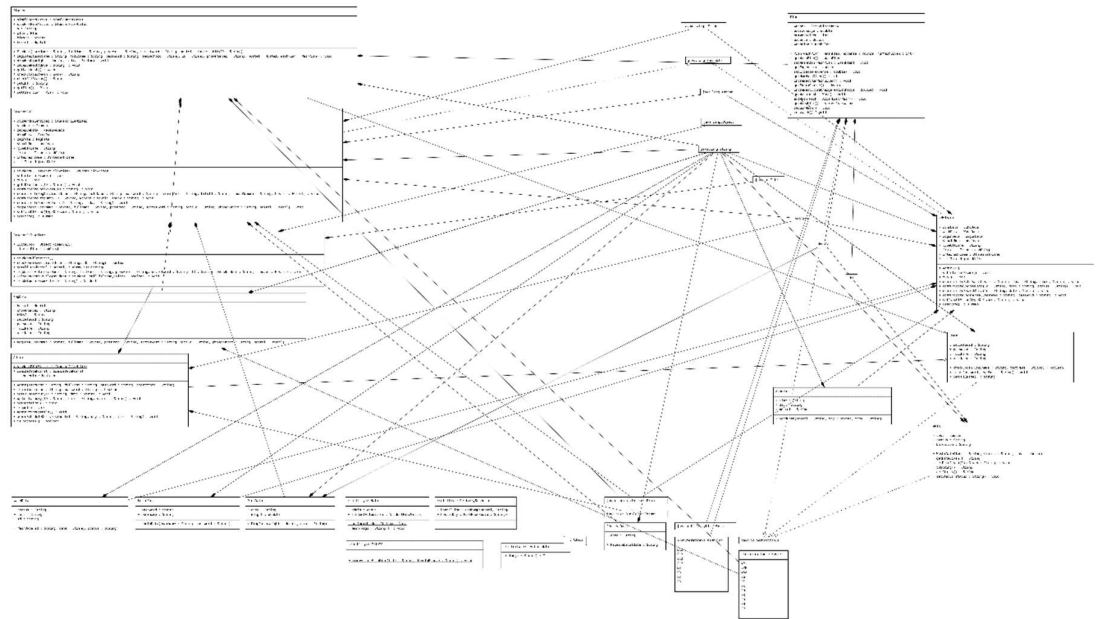
4. Interface Segregation:

Our program has a generic interface Actionable, which acts as an argument for the class SwingSingleInput_GUI. It only has a String input, and therefore does not require any further segregation.

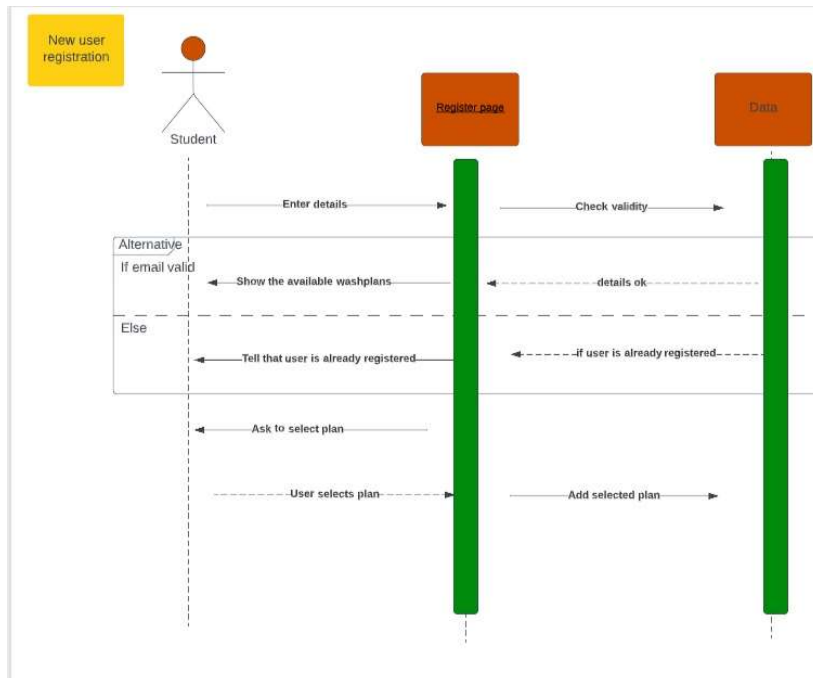
5. Dependency Inversion:

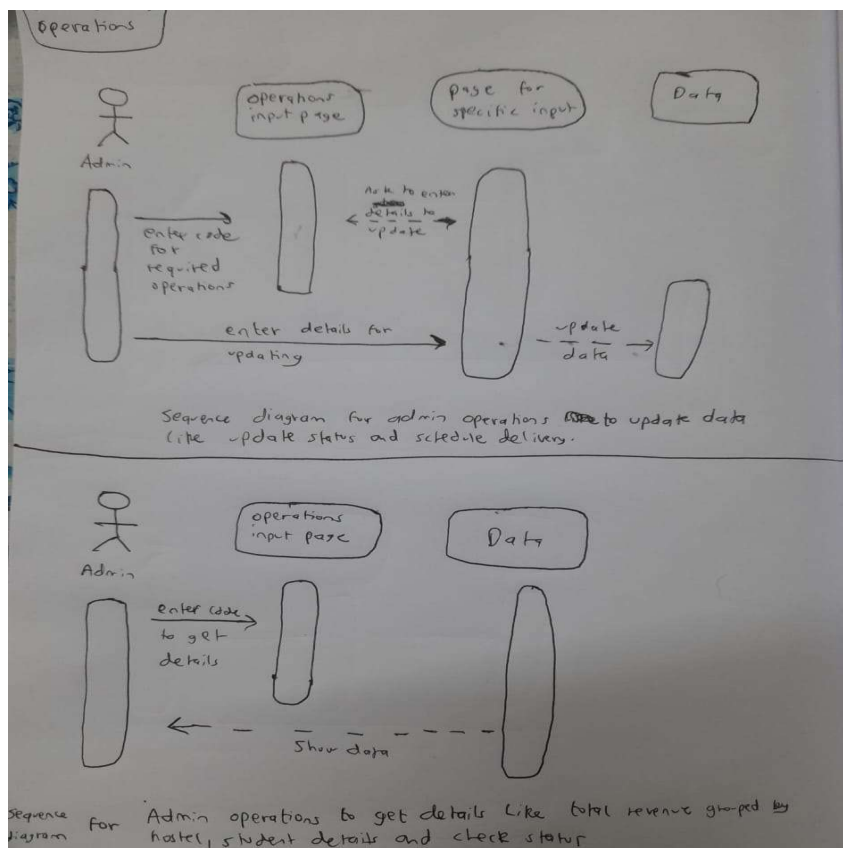
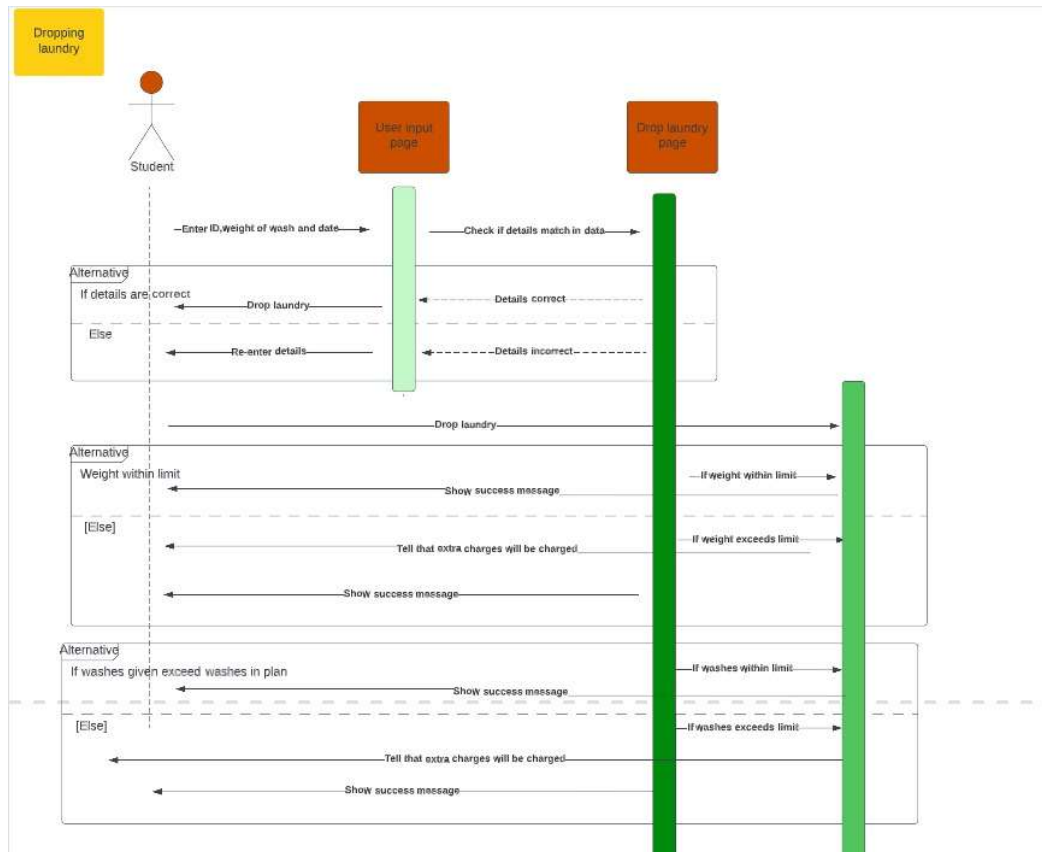
The classes in our code have extended the abstract User class, and other interfaces like Serializable and Runnable. In this way, the code depends on abstraction rather than concrete classes.

UML Class Diagram*:

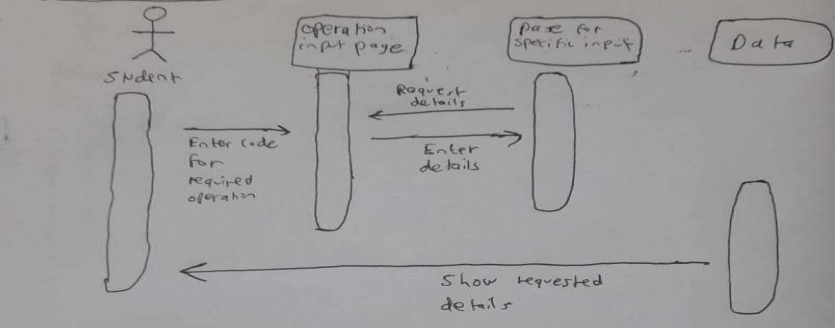


Sequence diagrams*:

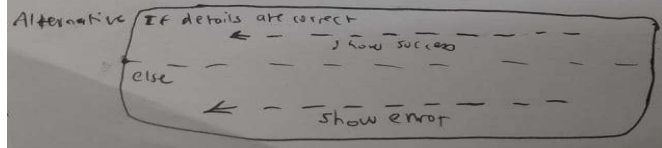




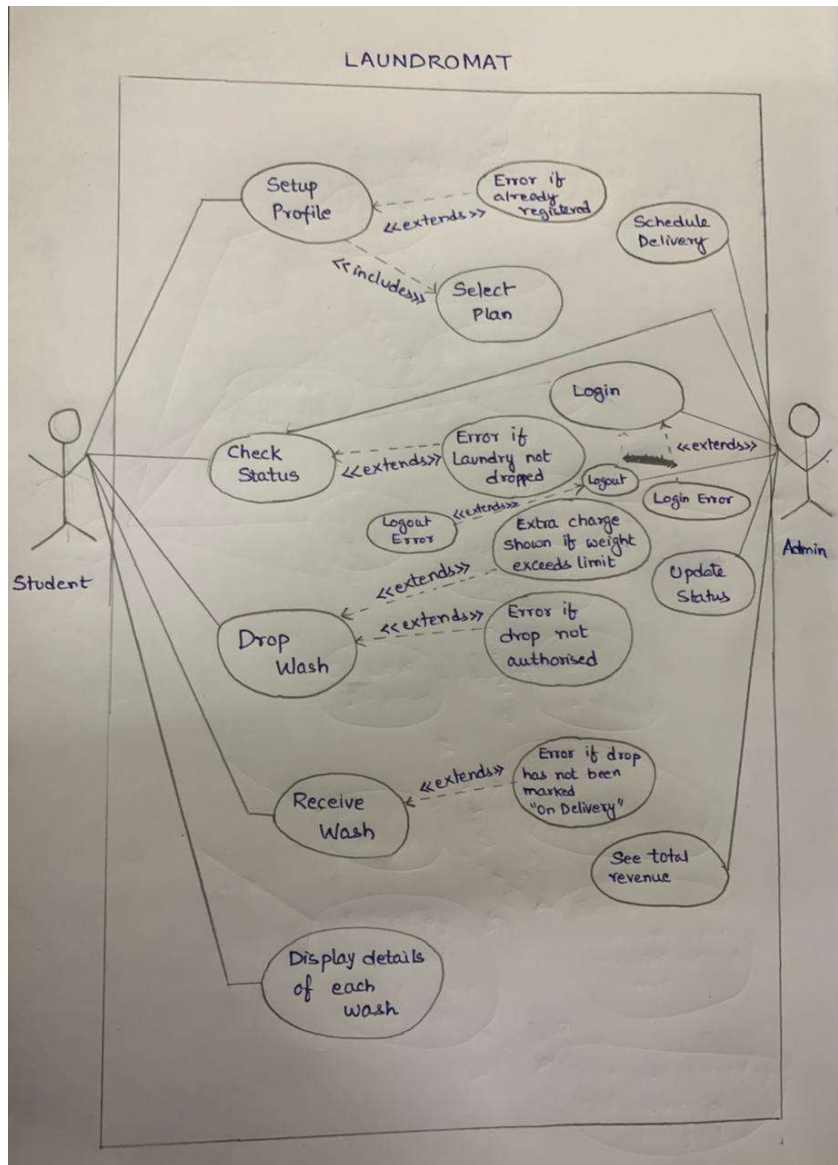
Other operations like
check status of last wash,
print details of all washes
and receive laundry



Admin login



Use Case Diagram*:



*All the UML diagram files are provided separately in the google drive