AIRPORT MANAGEMENT SYSTEM Code Analysis

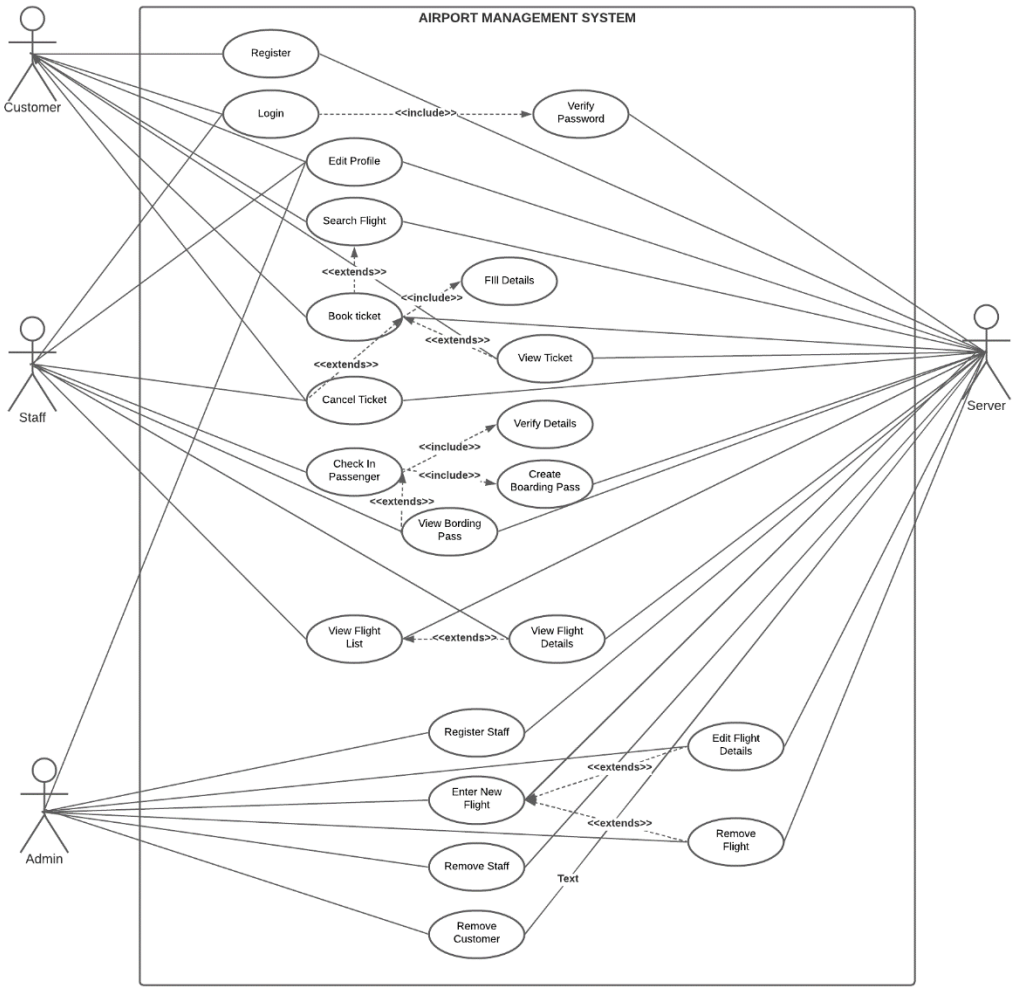
GROUP -90 Project 13

Sujay Patni 2019B3A70575P

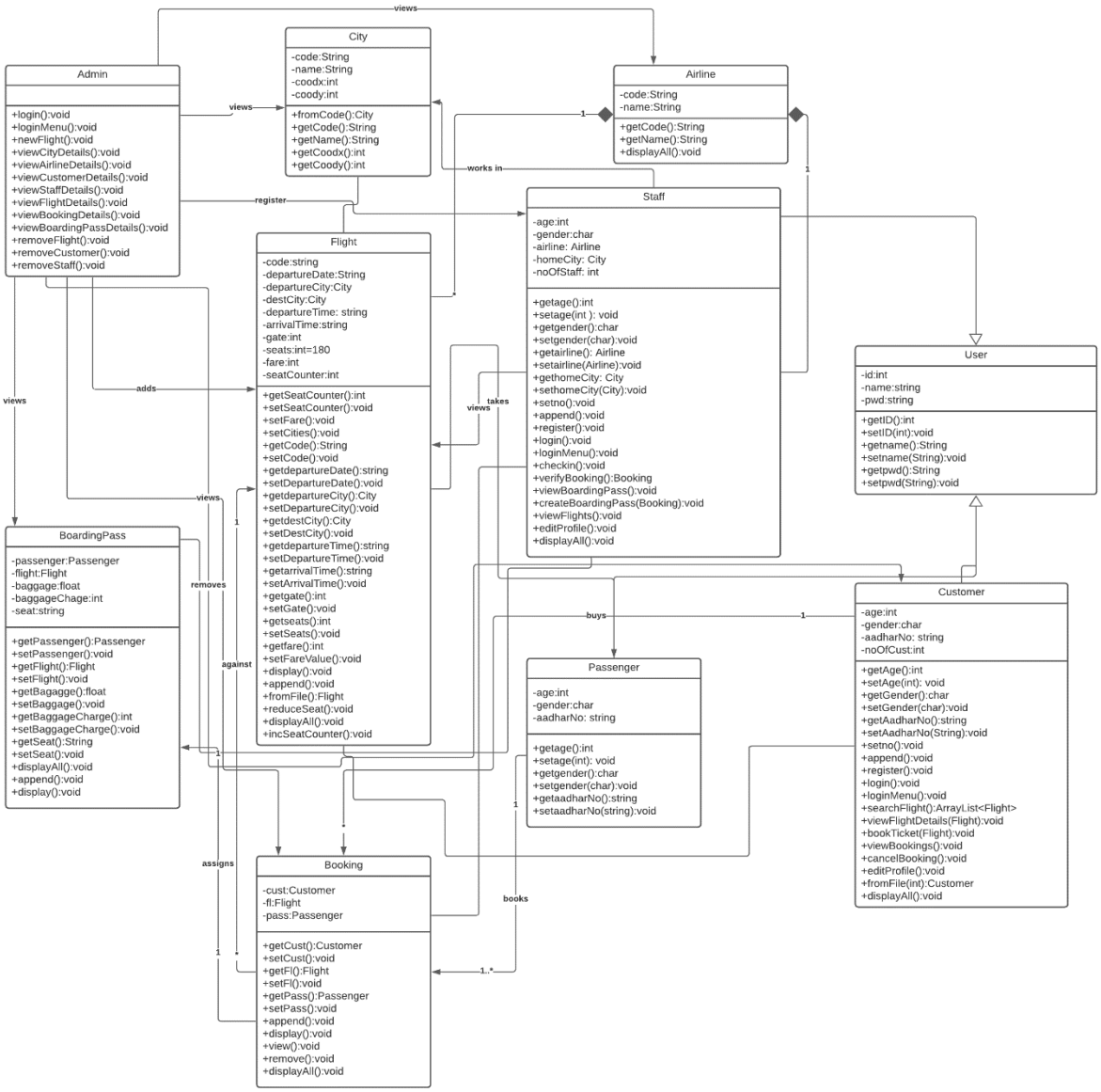
Vansh Chhabra 2019B1A71039P

UML Diagrams:

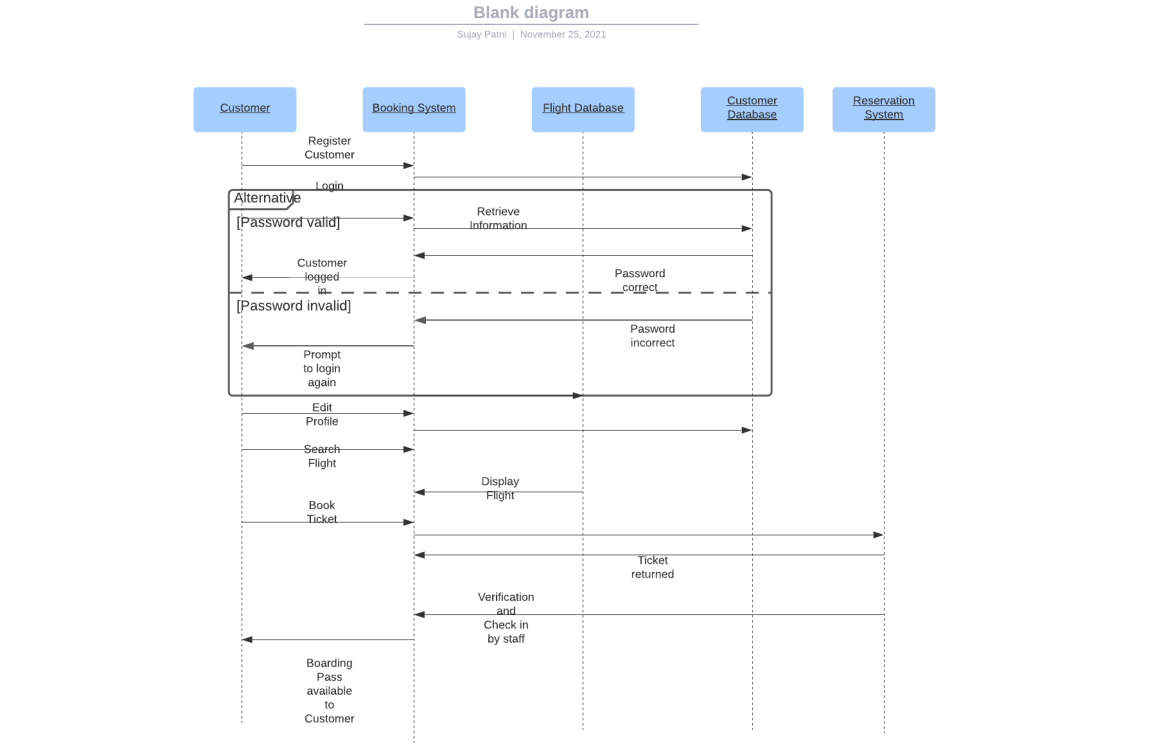
Use Case Diagram:

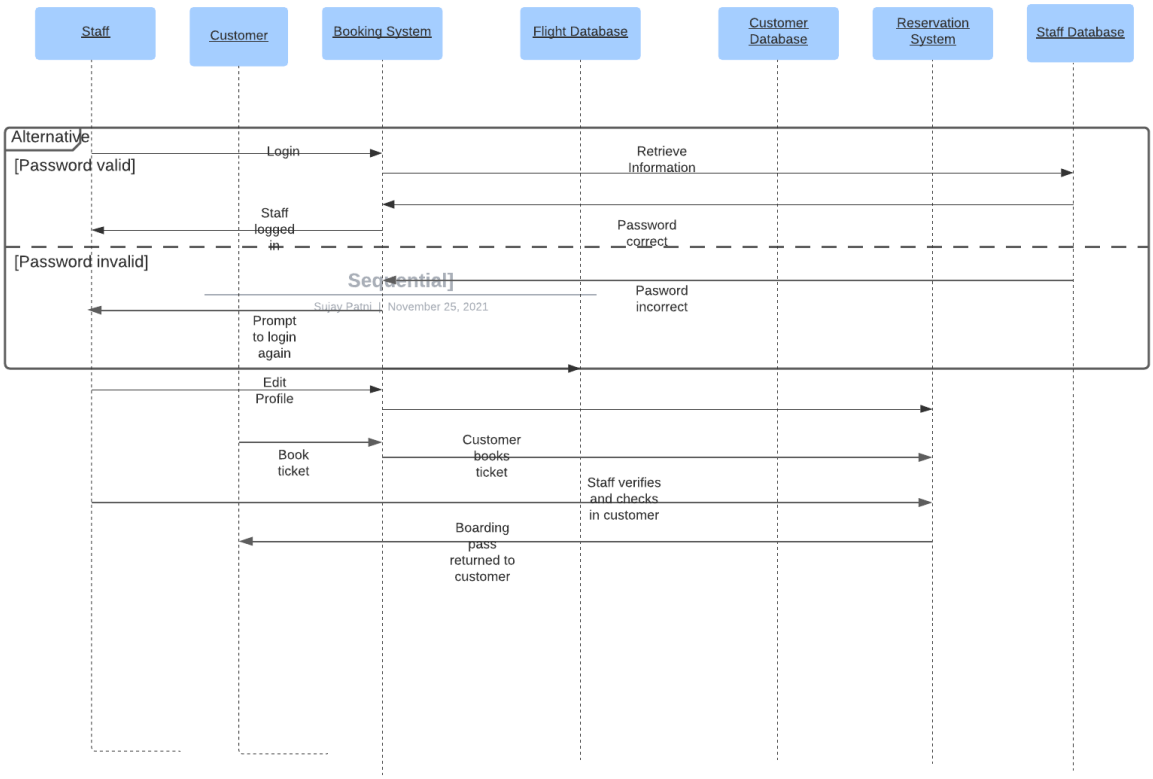


Class Diagram:



Sequential Diagrams:





OOP Principles Analysis:

1. Encapsulate what varies:

Methods that have different implementations in some classes and do not occur in other classes can be encapsulated in separate classes/interfaces as they vary in occurrence and implementation in different classes. Then the classes can inherit/implement them as required.

Such methods in our code are:

login(): Admin, Staff, Customer

loginMenu(): Admin, Staff, Customer

displayAll(): Airline, Staff, Flight, BoardingPass, Customer, Booking

display(): Flight, BoardingPass, Booking

append(): Staff, Flight, BoardingPass, Customer, Booking

getCode(): City, Airline, Flight

getName(): City, Airline, User

fromFile(): Customer, Flight

getAge(): Staff, Customer, Passenger

setAge(): Staff, Customer, Passenger

getGender(): Staff, Customer, Passenger

setGender(): Staff, Customer, Passenger

getAadharNo(): Customer, Passenger

setAadharNo(): Customer, Passenger

editProfile(): Customer, Staff

We could have used separate classes/interfaces for these and inherit from them but instead created same function again and again in all classes.

2. Favour composition over Inheritance

Our code has lots of examples favouring composition over inheritance (The principle that classes should achieve polymorphic behavior and code reuse by their composition, i.e., by containing instances of other classes that implement the desired functionality rather than inheritance from a base or parent class).

For example in the class BoardingPass, data members passenger and flight of types Passenger and Flight respectively have been used for implementing functionality. Instances of other classes have been used in this class.

The fromCode() method of City class returns the City class object as it is used to look up a particular city from the city database.

The Flight class implements the data members with their respective types: City departureCity, City destCity that are used to get the departure and destination cities.

The Booking Class implements Customer cust, Flight fl, and Passenger pass to get the respective customer, flight, and passenger information.

The Staff class implements Airline airline and City homeCity to get the respective airline and home city information.

The Customer class implements the searchFlight() method that returns a list of flights using an ArrayList containing references of the Flight class. The viewFlightDetails() and bookTicket() functions also take reference of the Flight class as an argument.

3. Depend on abstraction, do not depend on concrete classes

In our code we have not used an interface or abstract class anywhere. Using abstract classes or interfaces can prove to be very beneficial as we can exploit polymorphism by programming to a supertype so that the actual runtime object is not locked into the code.

For example, the User class could have been made abstract. Customer, Passenger, and Staff are the subclasses of the User class. If the User class would have been made abstract, then creating its object would not be possible and it would be better than the current implementation as currently a User object can be created which would not make any sense. Objects of the Customer, Passenger, and Staff classes have use but objects of the User class do not have any use.

Also, the setID() method of the User class is called in different ways in all three subclasses. If setID() would have been an abstract method, it would have to be defined separately in all three subclasses and defining them separately would have made more sense.

4. Strive for loose coupling between objects that interact

   Loose coupling is important as it promotes single responsibility and separation of concerns. If errors arise in one class/object, it should not directly affect the class/object it is linked to.

   In our code, we could have used interfaces so that the communication between classes happens through interfaces instead of concrete classes.

5. Classes should be open for extension and closed for modification

   Most of our classes follow this principle as after being coded they have been set and modification is not required unless a new feature has to be added

   Inheritance has been used so that the child classes can reuse the code of the parent class. For example, User class has been implemented as the parent class and Customer, Passenger, and Staff as its child classes.

   Using composition also helps in promoting this principle as other classes are being referred to by their objects. There is no direct use of the code of other classes in the class that refers to them. This reduces the chance of errors to occur.

Design Pattern

According to the UML class diagram attached in this same document, any particular design pattern has not been followed completely. In our code we have not used any interface or abstract class, instead we have used only concrete classes. Though, there is some resemblance with the **Factory Design pattern**. In such a pattern, a factory class consists of a factory method that returns an object considered to be generated by the factory. In our code, the methods that return objects are:

Method	**Return Type**	**Class**
getPassenger()	Passenger	BoardingPass
getFlight()	Flight	BoardingPass
fromCode()	City	City
getdepartureCity()	City	Flight
getdestCity()	City	Flight
fromFile()	Flight	Flight
getCust()	Customer	Booking
getFl()	Flight	Booking
getPass()	Passenger	Booking
getAirline()	Airline	Staff
gethomeCity()	City	Staff
verifyBooking()	Booking	Staff
searchFlight()	ArrayList<Flight>	Customer
fromFile()	Customer	Customer

Thus, in our code there are no such concrete factory methods that return created objects. In fact, most of such methods are getter functions used to return objects of these particular classes.

For example, we could use the Factory Design pattern for registering a user in our program. We would create a client class of the factory. The role of the factory is to produce (register) users. We would create a new class UserFactory and the client class would go through the UserFactory class to get instances of User.

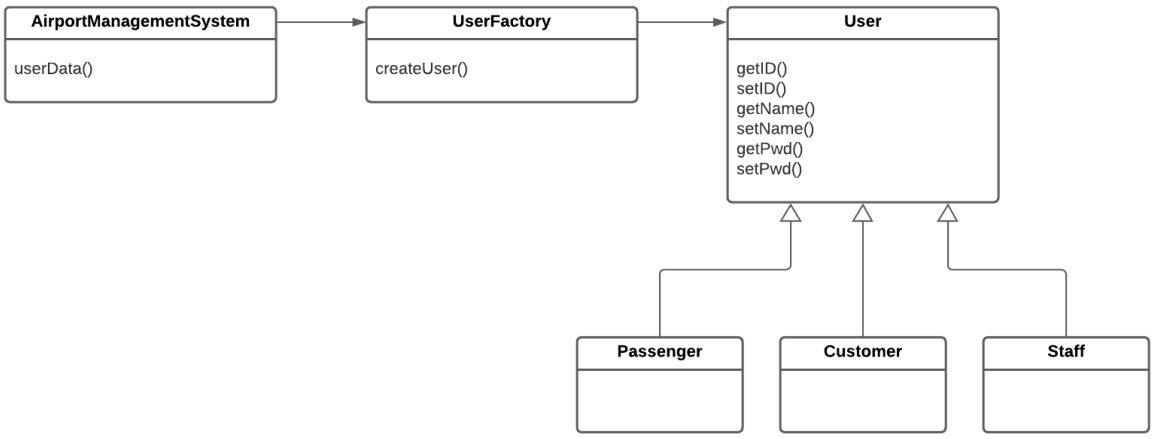
In UserFactory, we will create a static factory method createUser(). UserFactory is the factory where we create users. It is the only part of our application that refers to concrete User

classes. We will have an abstract User class having some implementations that can be overridden and which will give the product of the factory: a User.

Passenger, Customer, and Staff classes will be our concrete products. Each product will extend the abstract User class, i.e., be a subclass of the superclass User and will be concrete. With these conditions, they can be created by the factory and handed back to the client.

The pseudocode (blueprint) for the “factory” would look as follows:

```
public class UserFactory {

    public User createUser(String type) {

        User user = null;

        if(type.equals("passenger")) {

            user = new Passenger();

        } else if (type.equals("customer")) {

            user = new Customer();

        } else if (type.equals("staff")) {

            user = new Staff();
        }

        return user;

    }

}
```



Overall, in the evaluation of code, the improvements apart from those mentioned above could be implementing more interfaces and abstract classes, as well as inheritance in terms of both concrete and abstract classes. We could have also interfaces to create flight, book flights and create Boarding Pass but chose a very direct raw approach.

Improvement/Future Work

1. We have not followed the OOP principles, thus registering that in our blueprint would make it a lot cleaner.
2. We have not followed any Design Pattern, following a design pattern would have improved the code a lot.
3. All the inputs taken from the users are case sensitive and format sensitive, thus typing something other than the exact expected input would not give the same results. Due to time constraints, we could not include these cases in our program.
4. We have not been able to do exception handling in the program due to time constraint, thus there might be a few border cases where the program may faulter (not found any yet). Further, giving the program a wrong input (String instead of Int) would give an exception and close the program. Instead, asking the user to rewrite the input would have been more user-friendly but we couldn't include it.
5. The program assigns the seat to a passenger itself instead of asking preference, which would have been more realistic.
6. Dividing the airline into Economy, Business and First Class and charging different for different Class would have been more realistic.
7. Adding GUI to the program.
8. Creating a better system for Flight Management as in our code, Flight is actually the Flight Instance and is different for each date. Creating a common Flight Class that had function to take in different dates when the plane flies would have been much better but it was really hard to implement and we were unable to do it in due time.
9. Charging fare based on the date the ticket is booked.