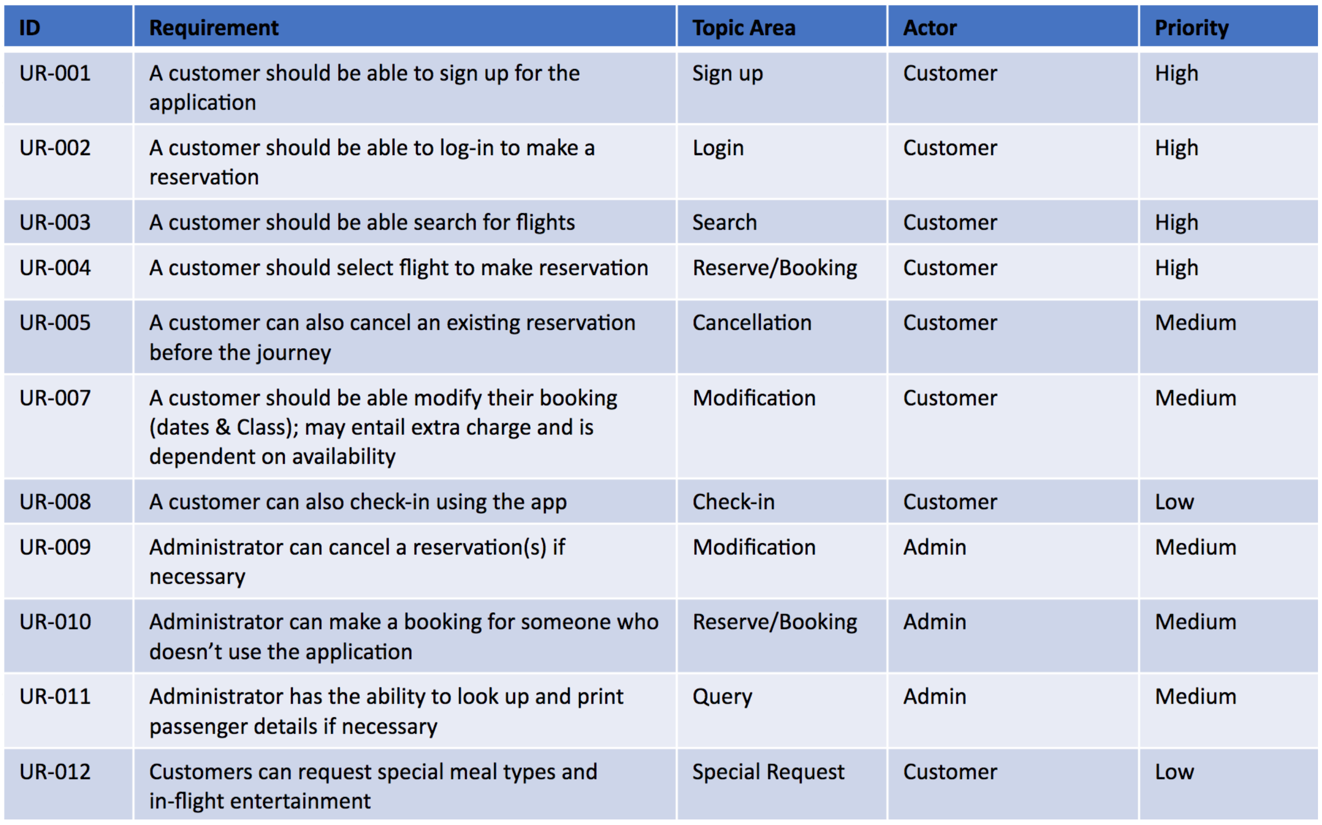
Flight Reservation Application

Attaluri Laxmi Naga Santosh, Saikrishna Jaliparthy, Shruthi Sukumar

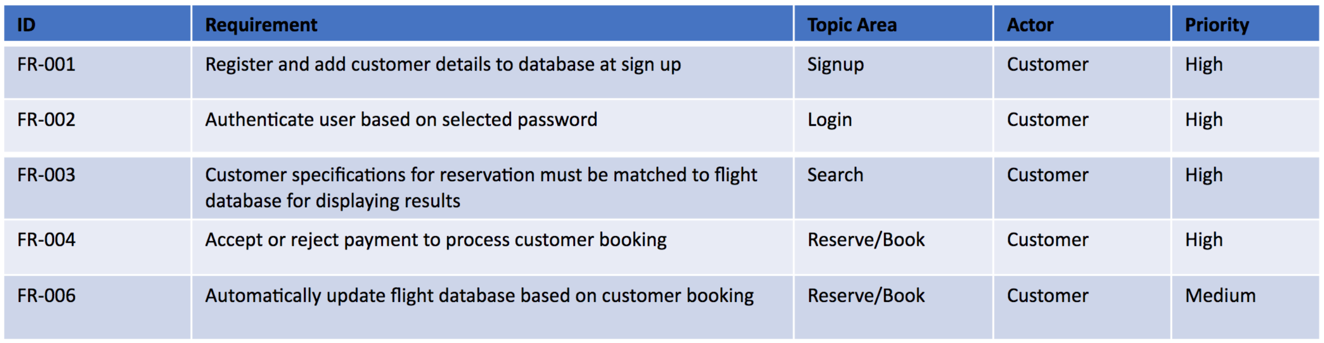
Project Summary: This application enables users to reserve an airplane booking for travel on specified date and between specified airports. Customers can input details of their desired itinerary and pick a one-way or round trip desired flight. Then the reservation is confirmed by an online payment by means of debit or credit card. Customers are initimated by email through the system.

1. The following tables indicate the features/requirements that were completed from the ones listed in our Project Part 2 submission.



ID	Requirement	Topic Area	Actor	Priority
UR-001	A customer should be able to sign up for the application	Sign up	Customer	High
UR-002	A customer should be able to log-in to make a reservation	Login	Customer	High
UR-003	A customer should be able search for flights	Search	Customer	High
UR-004	A customer should select flight to make reservation	Reserve/Booking	Customer	High
UR-005	A customer can also cancel an existing reservation before the journey	Cancellation	Customer	Medium
UR-007	A customer should be able modify their booking (dates & Class); may entail extra charge and is dependent on availability	Modification	Customer	Medium
UR-008	A customer can also check-in using the app	Check-in	Customer	Low
UR-009	Administrator can cancel a reservation(s) if necessary	Modification	Admin	Medium
UR-010	Administrator can make a booking for someone who doesn't use the application	Reserve/Booking	Admin	Medium
UR-011	Administrator has the ability to look up and print passenger details if necessary	Query	Admin	Medium
UR-012	Customers can request special meal types and in-flight entertainment	Special Request	Customer	Low

Figure 1: User Requirements from Part 2 that were succesfully implemented

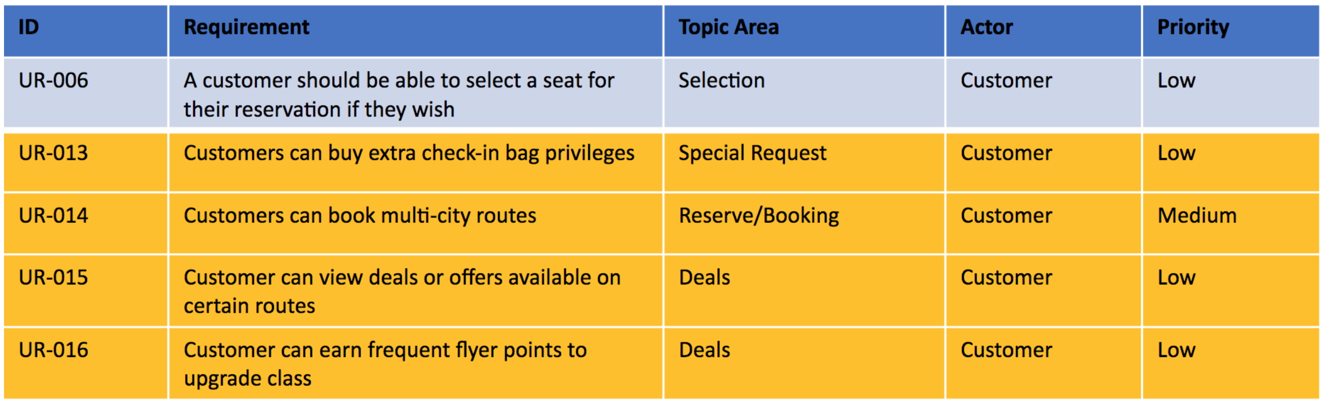


ID	Requirement	Topic Area	Actor	Priority
FR-001	Register and add customer details to database at sign up	Signup	Customer	High
FR-002	Authenticate user based on selected password	Login	Customer	High
FR-003	Customer specifications for reservation must be matched to flight database for displaying results	Search	Customer	High
FR-004	Accept or reject payment to process customer booking	Reserve/Book	Customer	High
FR-006	Automatically update flight database based on customer booking	Reserve/Book	Customer	Medium

Figure 2: Functional Requirements from Part 2 that were succesfully implemented

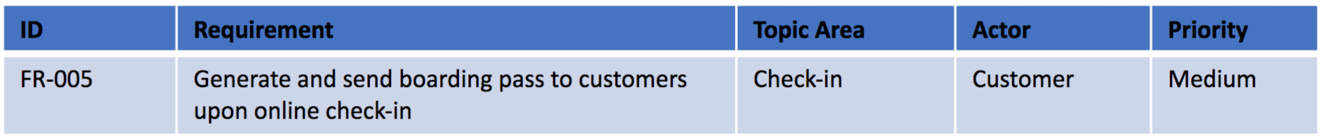
These labels and ID numbers are kept consistent with the report submitted for Part 2 to make comparisons easier. Most of the High priority items were implemented to get a basic model of the application up and running. However, as in the case UR-006 to UR-011 as well as FR-006, we did also implement low and medium priority items on our list.

2. Listed below are features that weren't implemented in our running application.



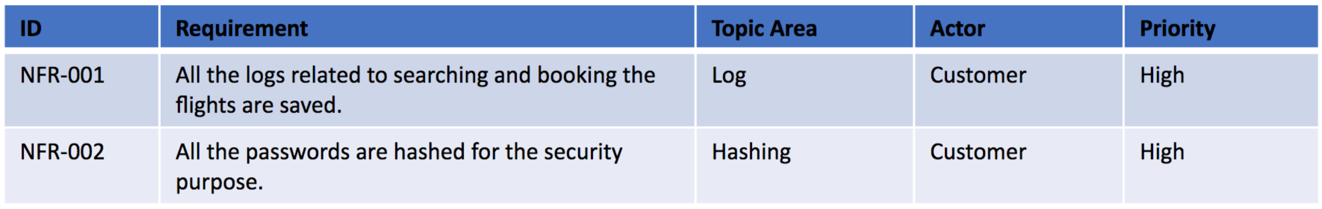
ID	Requirement	Topic Area	Actor	Priority
UR-006	A customer should be able to select a seat for their reservation if they wish	Selection	Customer	Low
UR-013	Customers can buy extra check-in bag privileges	Special Request	Customer	Low
UR-014	Customers can book multi-city routes	Reserve/Booking	Customer	Medium
UR-015	Customer can view deals or offers available on certain routes	Deals	Customer	Low
UR-016	Customer can earn frequent flyer points to upgrade class	Deals	Customer	Low

Figure 3: User Requirements Not Implemented



ID	Requirement	Topic Area	Actor	Priority
FR-005	Generate and send boarding pass to customers upon online check-in	Check-in	Customer	Medium

Figure 4: Functional Requirements Not Implemented

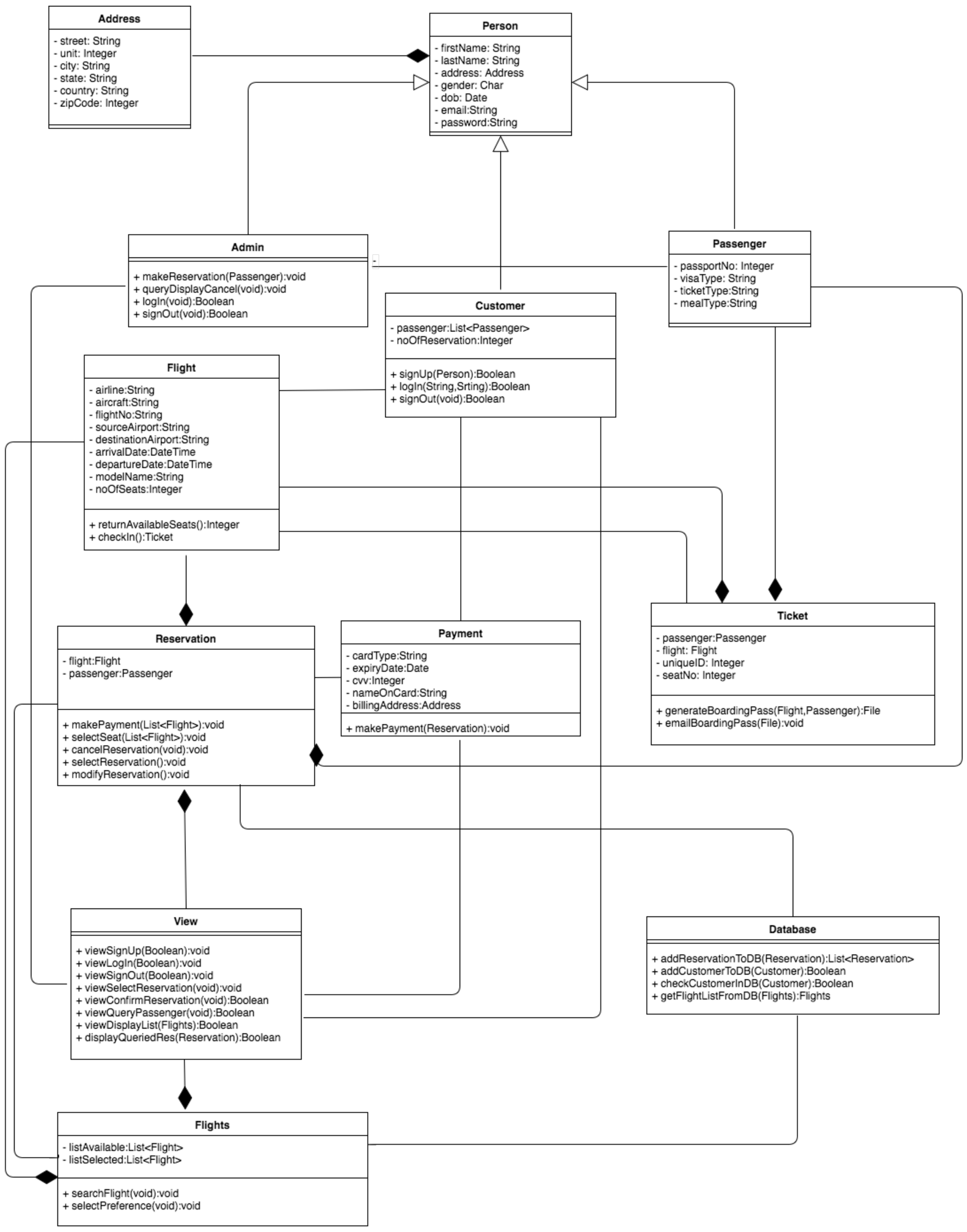


ID	Requirement	Topic Area	Actor	Priority
NFR-001	All the logs related to searching and booking the flights are saved.	Log	Customer	High
NFR-002	All the passwords are hashed for the security purpose.	Hashing	Customer	High

Figure 5: Non-Functional Requirements Not Implemented

As stated before, the table titles are kept consistent with Part 2 submission to facilitate easy comparisons. A lot of the user requirements that were not implemented were stretch requirements. These were requirements that were introduced in the hopes to make the application more versatile in terms of what can be done, like adding multi-city routes to their itineraries and collecting travel points which would give them travel deals. We haven't implemented hashing of passwords. These are high-priority requirements/features with respect to proper functioning of the application if it were ever to be deployed. However, being relatively easy features to implement, in the interest of time we chose to implement features more in line with the objectives of the class, namely the re-factoring to implement design patterns.

3. Shown below are the initial class diagram we began with (submitted in Part 2) and the class diagram of what we ended up implementing.



Address
- street: String
- unit: Integer
- city: String
- state: String
- country: String
- zipCode: Integer
Person
- firstName: String
- lastName: String
- address: Address
- gender: Char
- dob: Date
- email:String
- password:String
Admin
+ makeReservation(Passenger):void
+ queryDisplayCancel(void):void
+ logIn(void):Boolean
+ signOut(void):Boolean
Passenger
- passportNo: Integer
- visaType: String
- ticketType:String
- mealType:String
Customer
- passenger:List<Passenger>
- noOfReservation:Integer
+ signUp(Person):Boolean
+ logIn(String,Srting):Boolean
+ signOut(void):Boolean
Flight
- airline:String
- aircraft:String
- flightNo:String
- sourceAirport:String
- destinationAirport:String
- arrivalDate:DateTime
- departureDate:DateTime
- modelName:String
- noOfSeats:Integer
+ returnAvailableSeats():Integer
+ checkIn():Ticket
Reservation
- flight:Flight
- passenger:Passenger
+ makePayment(List<Flight>):void
+ selectSeat(List<Flight>):void
+ cancelReservation(void):void
+ selectReservation():void
+ modifyReservation():void
Payment
- cardType:String
- expiryDate:Date
- cvv:Integer
- nameOnCard:String
- billingAddress:Address
+ makePayment(Reservation):void
Ticket
- passenger:Passenger
- flight: Flight
- uniqueID: Integer
- seatNo: Integer
+ generateBoardingPass(Flight,Passenger):File
+ emailBoardingPass(File):void
View
+ viewSignUp(Boolean):void
+ viewLogIn(Boolean):void
+ viewSignOut(Boolean):void
+ viewSelectReservation(void):void
+ viewConfirmReservation(void):Boolean
+ viewQueryPassenger(void):Boolean
+ viewDisplayList(Flights):Boolean
+ displayQueriedRes(Reservation):Boolean
Database
+ addReservationToDB(Reservation):List<Reservation>
+ addCustomerToDB(Customer):Boolean
+ checkCustomerInDB(Customer):Boolean
+ getFlightListFromDB(Flights):Flights
Flights
- listAvailable:List<Flight>
- listSelected:List<Flight>
+ searchFlight(void):void
+ selectPreference(void):void

Figure 6: Class Diagram submitted in part 2

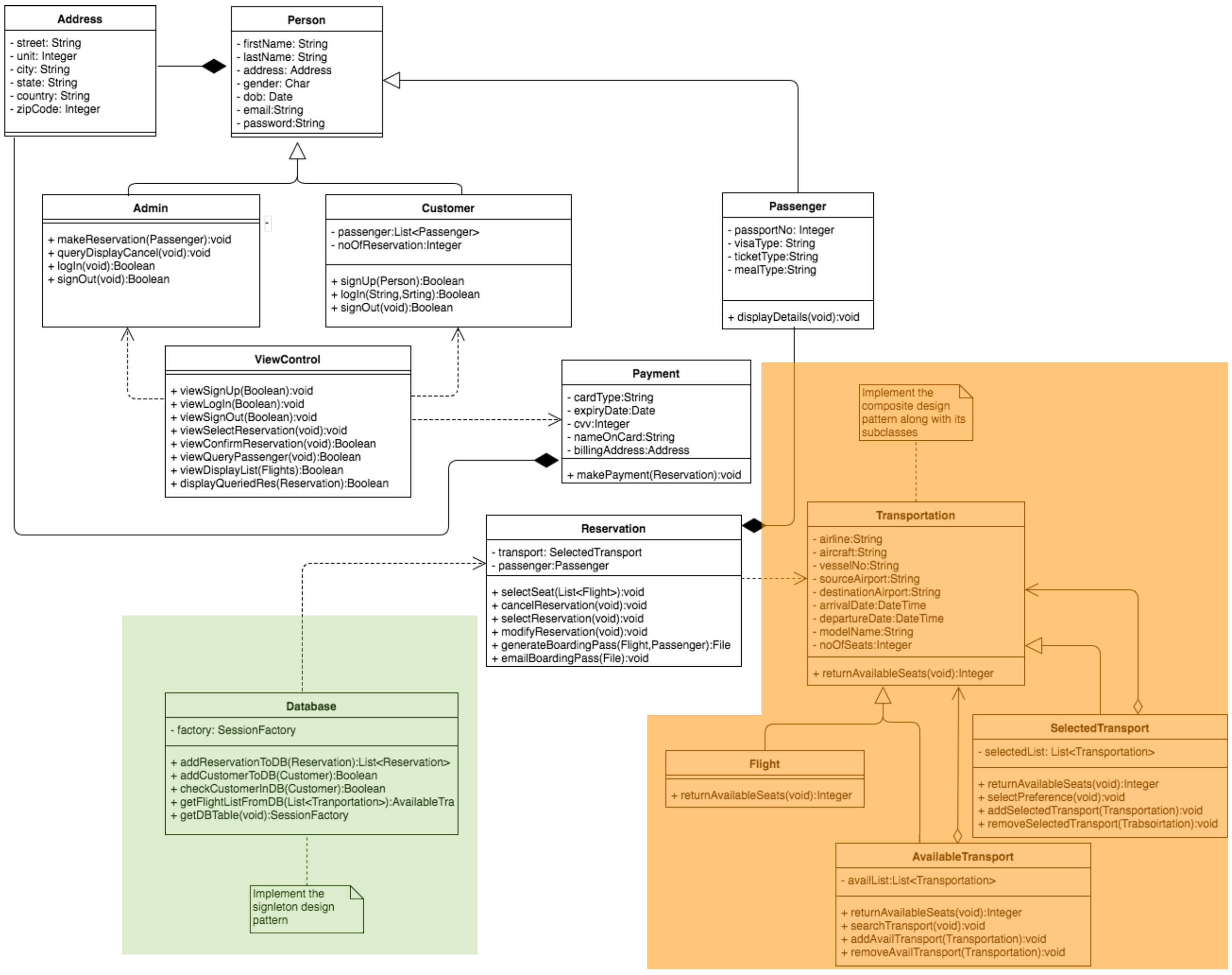

Address
- street: String
- unit: Integer
- city: String
- state: String
- country: String
- zipCode: Integer
Person
- firstName: String
- lastName: String
- address: Address
- gender: Char
- dob: Date
- email:String
- password:String
Admin
+ makeReservation(Passenger):void
+ queryDisplayCancel(void):void
+ logIn(void):Boolean
+ signOut(void):Boolean
Customer
- passenger:List<Passenger>
- noOfReservation:Integer
+ signUp(Person):Boolean
+ logIn(String,Srting):Boolean
+ signOut(void):Boolean
Passenger
- passportNo: Integer
- visaType: String
- ticketType:String
- mealType:String
+ displayDetails(void):void
ViewControl
+ viewSignUp(Boolean):void
+ viewLogIn(Boolean):void
+ viewSignOut(Boolean):void
+ viewSelectReservation(void):void
+ viewConfirmReservation(void):Boolean
+ viewQueryPassenger(void):Boolean
+ viewDisplayList(Flights):Boolean
+ displayQueriedRes(Reservation):Boolean
Payment
- cardType:String
- expiryDate:Date
- cvv:Integer
- nameOnCard:String
- billingAddress:Address
+ makePayment(Reservation):void
Implement the composite design pattern along with its subclasses
Reservation
- transport: SelectedTransport
- passenger:Passenger
+ selectSeat(List<Flight>):void
+ cancelReservation(void):void
+ selectReservation(void):void
+ modifyReservation(void):void
+ generateBoardingPass(Flight,Passenger):File
+ emailBoardingPass(File):void
Transportation
- airline:String
- aircraft:String
- vesselNo:String
- sourceAirport:String
- destinationAirport:String
- arrivalDate:DateTime
- departureDate:DateTime
- modelName:String
- noOfSeats:Integer
+ returnAvailableSeats(void):Integer
Database
- factory: SessionFactory
+ addReservationToDB(Reservation):List<Reservation>
+ addCustomerToDB(Customer):Boolean
+ checkCustomerInDB(Customer):Boolean
+ getFlightListFromDB(List<Tranportation>):AvailableTra
+ getDBTable(void):SessionFactory
Implement the signleton design pattern
Flight
+ returnAvailableSeats(void):Integer
SelectedTransport
- selectedList: List<Transportation>
+ returnAvailableSeats(void):Integer
+ selectPreference(void):void
+ addSelectedTransport(Transportation):void
+ removeSelectedTransport(Trabsoirtation):void
AvailableTransport
- availList:List<Transportation>
+ returnAvailableSeats(void):Integer
+ searchTransport(void):void
+ addAvailTransport(Transportation):void
+ removeAvailTransport(Transportation):void

Figure 7: Final Class Diagram Similar to what was submitted in part 3

Obviously, the biggest difference between the class diagram in Figure 6 and the one in Figure 7 is the use of design patterns. There are also changes to some other classes to better reflect the function they carry out. This happens with ‘View’ in Figure 6 which has been changed to ‘ViewControl’ in Figure 7 to better reflect that it performs view as well as controller duties. We also eliminated the ‘Ticket’ in the first diagram and added its attributes to the ‘Reservation’ in Figure 7.Some of the mistaken connections, which were also pointed out in the Part 2 submission, were corrected.

4. Design patterns:
The singleton design pattern was applied to the database class (highlighted in green in Figure 7) to ensure multiple sessions while working with hibernate are not created. The singleton design pattern here isn't used in the conventional way, as in, we are not attempting to create only a single instance of the database class, but we are ensuring that it is used to ensure only one session exists when working with the hibernate tool.

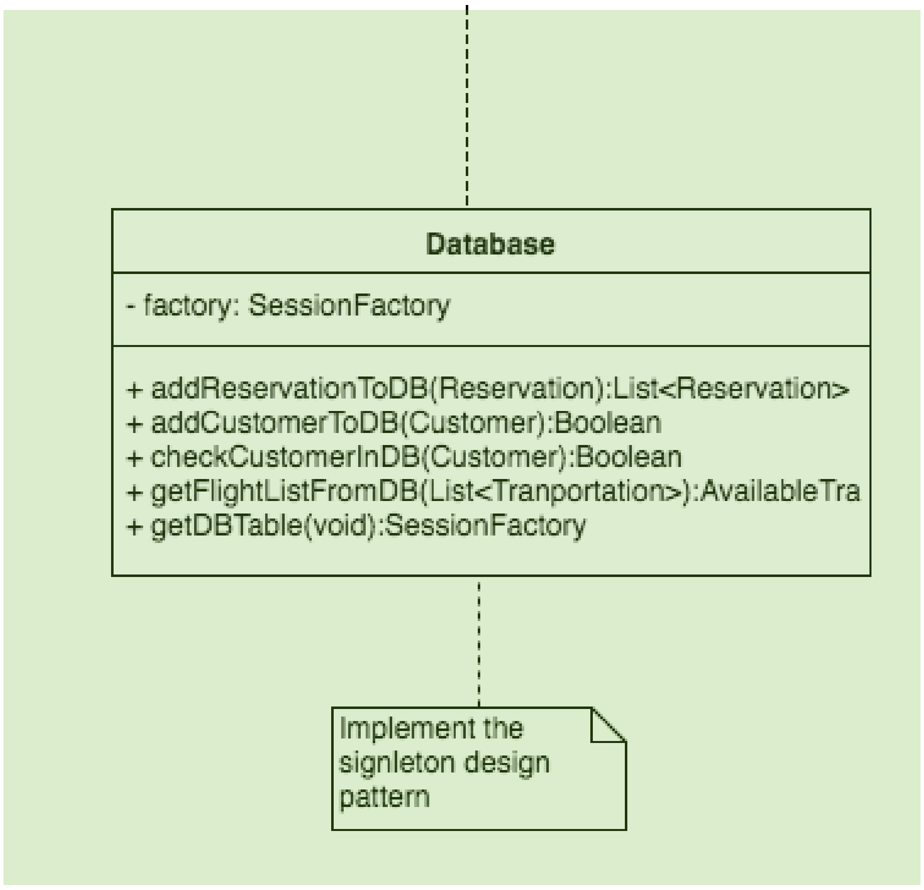
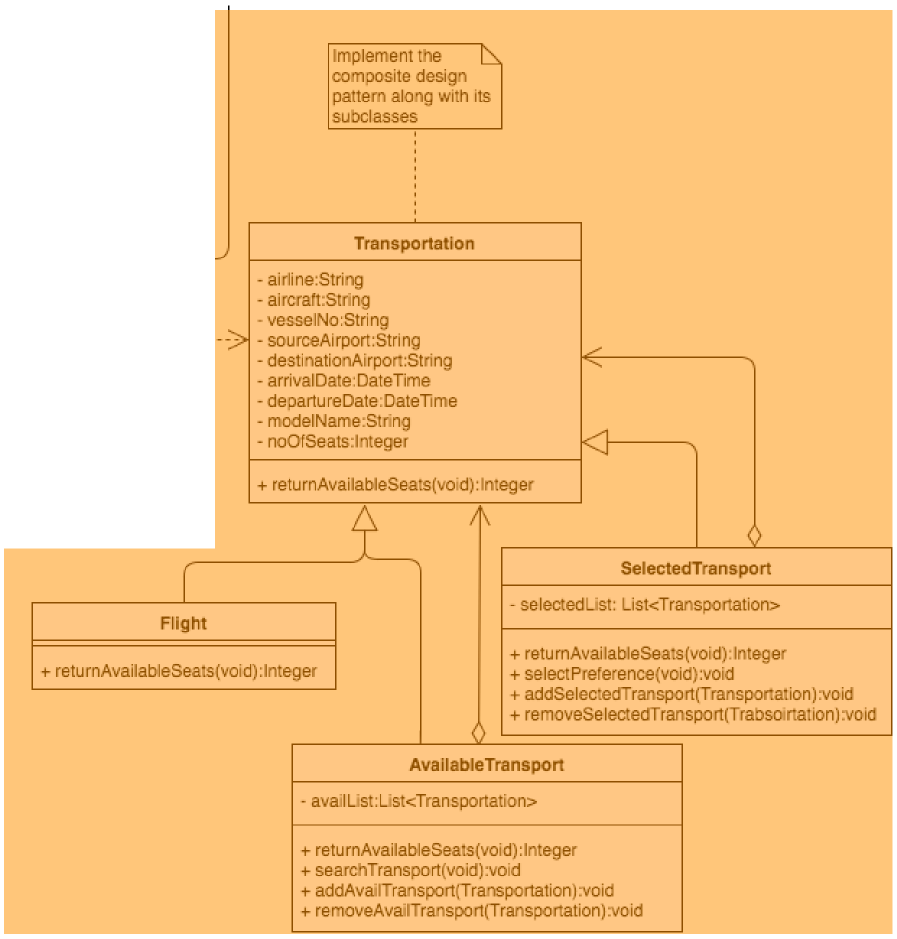

Database
- factory: SessionFactory
+ addReservationToDB(Reservation):List<Reservation>
+ addCustomerToDB(Customer):Boolean
+ checkCustomerInDB(Customer):Boolean
+ getFlightListFromDB(List<Tranportation>):AvailableTra
+ getDBTable(void):SessionFactory
Implement the signleton design pattern

Figure 8: Singleton Design Pattern

The other big change is to remove two classes 'Flight' and 'Flights' to use compositions more intelligently by means of the Composite design pattern (highlighted in orange in Figure 7). The structure of the Composite design pattern has been implemented by us with the hope that it would extend it to implementing it in the future when we have to add more types of transportation.



Implement the composite design pattern along with its subclasses
Transportation
- airline:String
- aircraft:String
- vesselNo:String
- sourceAirport:String
- destinationAirport:String
- arrivalDate:DateTime
- departureDate:DateTime
- modelName:String
- noOfSeats:Integer
+ returnAvailableSeats(void):Integer
Flight
+ returnAvailableSeats(void):Integer
SelectedTransport
- selectedList: List<Transportation>
+ returnAvailableSeats(void):Integer
+ selectPreference(void):void
+ addSelectedTransport(Transportation):void
+ removeSelectedTransport(Trabsoirtation):void
AvailableTransport
- availList:List<Transportation>
+ returnAvailableSeats(void):Integer
+ searchTransport(void):void
+ addAvailTransport(Transportation):void
+ removeAvailTransport(Transportation):void

Figure 9: Composite Design Pattern

Future Work:

We could have also implemented the Decorator design pattern to more intelligently perform searches for different transportation mediums. However our system was designed to include very simple round trip flight selections for which the decorator design pattern wasn't really necessary to use. Another design pattern that could have been applied to make the system efficient was the Facade pattern, however, the refactoring to include it would have resulted in more classes than it would have saved us.

5. The process of analysis and design of software systems:

Overall, the structure of the final project required the group to work in a team and envision how the software application would work before getting down to coding it. The existence of a unified language to communicate the workings of the system at different levels, from higher abstract levels in the activity diagram to implementational levels in the sequence and class diagrams outlined a clear path for coding the logic that needs to be done, while keeping in mind the ultimate goal of the project. The lessons we learned through the design principles helped improve our implementational abilities and they were enhanced by means of these 'off-the-shelf' recipes in the forms of design patterns that we could apply to our software design. The time and effort spent in design documentation before coding up the system shed light on how this process is fast expedited with little resource wasted on what the user needs by ironing out the details before the actual implementation. As an aside, the use of Hibernate, Spring MVC (not in our project) and Maven as well as tools like Eclipse were the useful skills that were picked up from this class over the course of this semester.