- **Team:**
  - Attaluri Laxmi Naga Santosh
  - Saikrishna Jaliparthy
  - Shruthi Sukumar
- **Title:** Flight Reservation Application
- **Refactoring Analysis:**
  On the whole, the refactoring exercise initiated a re-thinking of our original application design. Overall, we changed the structure to reduce coupling between classes in comparison to our previous class diagram. We removed a redundant class **Ticket** and incorporated the methods into the **Reservation** class. Figure 1 shows the original class diagram and Figure 3 shows our refactored design-based class diagram. Figure 2 depicts the re-arranged, cleaned up version of our original class diagram to make it more readable and to remove incosistencies based on the Part 2 submission feedback.

  Design Patterns:

  Based on the GoF Design Patterns textbooks as well as Prof. Boese's slides, we incorporated two of the main design patters in our software design. First, for database access we used the **Singleton** DP. The **Database** class in our class diagram acts as the access point from our code to the database which contains list of flights, customers and reservations. Using the **Singleton** design pattern, the **Database** class by means of the **getDBTable()** method, ensures that no more than one session is created. The following is the code snippet for the **Database** class:

```
class Database
{
private static SessionFactory factory;

public static synchronized SessionFactory getDBTable()
{
if(factory==null)
factory=new Configuration().configure().buildSessionFactory();
return factory;
}

}
```

  Another design pattern that was implemented in our design during refactoring was the **Composite** design pattern. The **Transportation** class is the base class for all possible transportations in an itinerary. Some airports include a chopper and bus in addition to flights. A group of flights to be listed in a search as well as the list of flights in the reservation are compositions of transportations that are available. The application of the **Composite** design patterns enables the groups of transport objects to be treated uniformly as a single flight (or bus or chopper). Additionally, using this design pattern makes it easy to add new classes for any additional transport medium in the future, without disturbing the base **Transportation** class. In our class diagram, we have used a "double" composition of transportation objects; one to represent the list of available flights from the search (or buses and choppers), and the other to represent the list of selected transportation in a reservation. The **Reservation** class acts as the *client* in this implementation of the **Composite** design pattern (Figure 3).

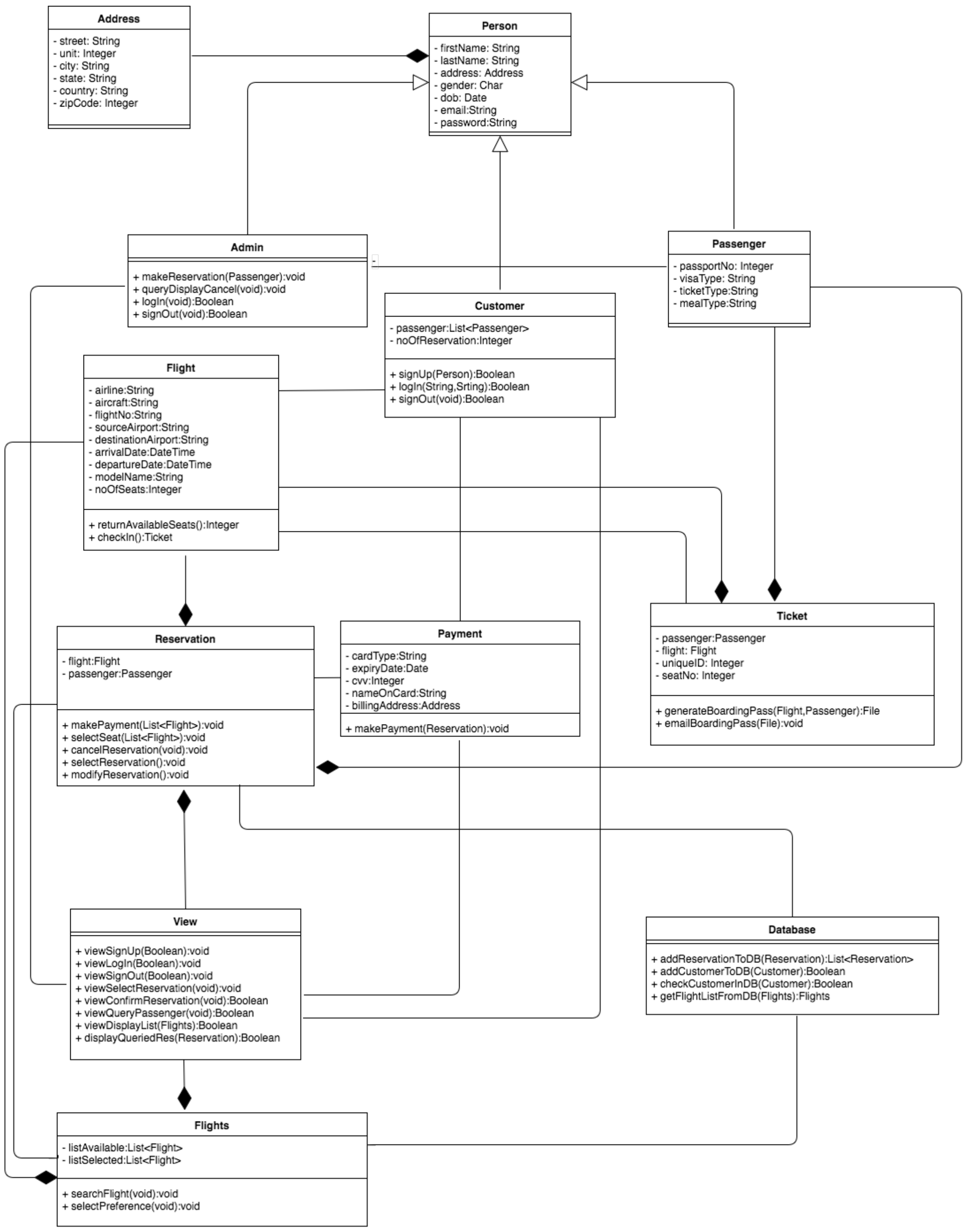
Address
- street: String
- unit: Integer
- city: String
- state: String
- country: String
- zipCode: Integer
Person
- firstName: String
- lastName: String
- address: Address
- gender: Char
- dob: Date
- email:String
- password:String
Admin
+ makeReservation(Passenger):void
+ queryDisplayCancel(void):void
+ 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 1: Original Class diagram: Shows compositions, inheritance and dependency relations

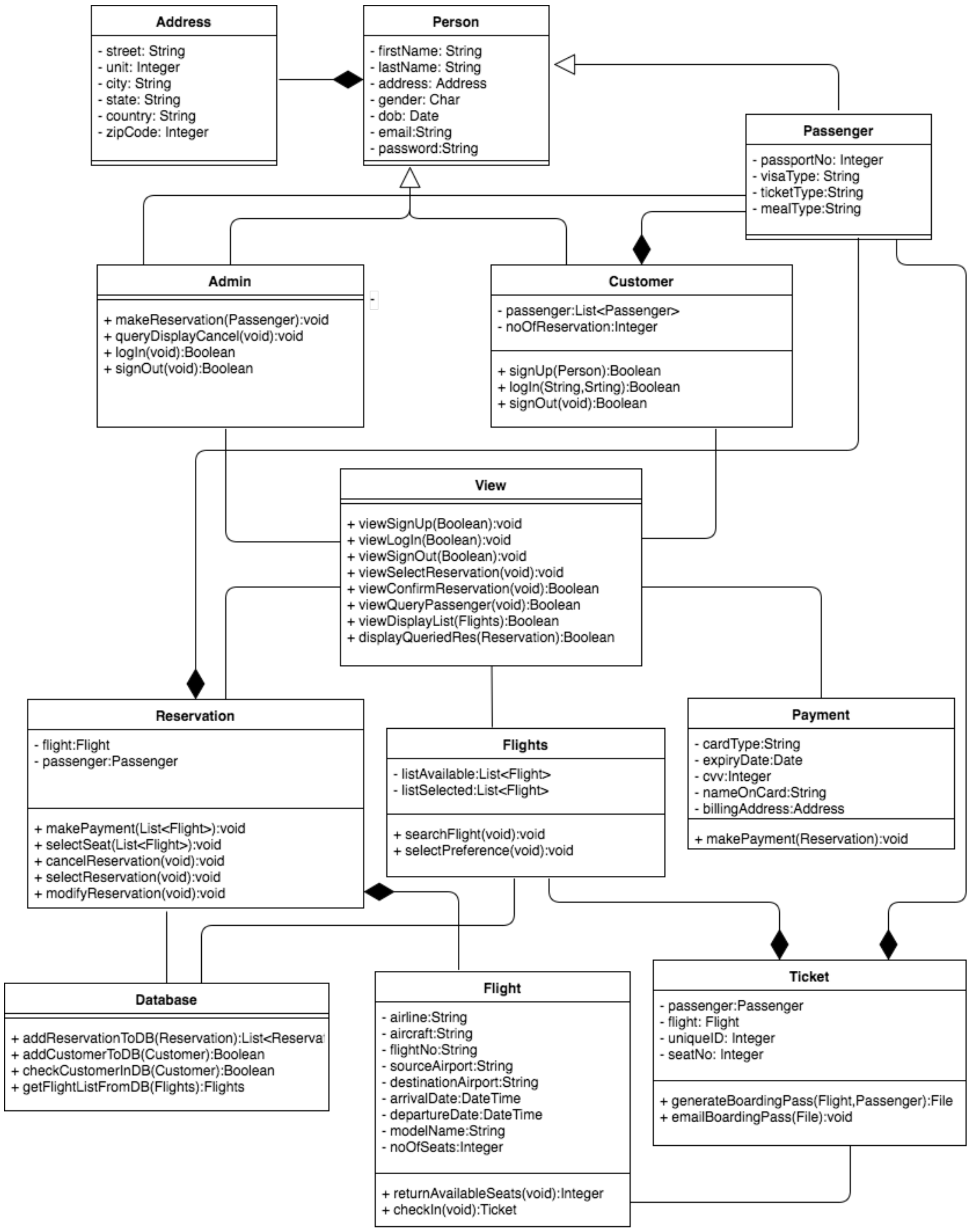
Address
- street: String
- unit: Integer
- city: String
- state: String
- country: String
- zipCode: Integer
Person
- firstName: String
- lastName: String
- address: Address
- gender: Char
- dob: Date
- email:String
- password:String
Passenger
- passportNo: Integer
- visaType: String
- ticketType:String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

Figure 2: Original Class diagram: Cleaned up version removing some of the inconsistencies noted in the part 2 feedback and in general making it more readable

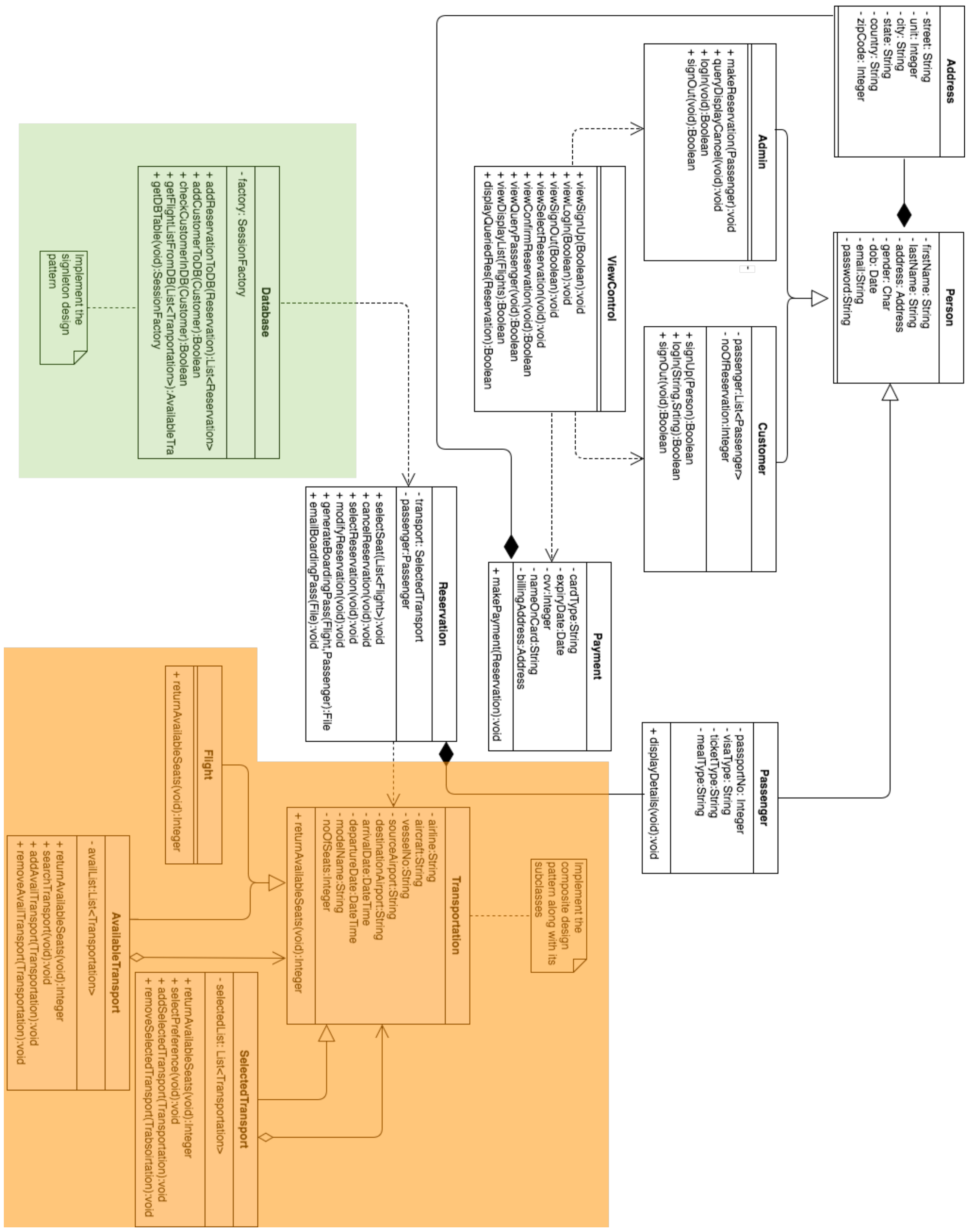
Address
- street: String
- unit: Integer
- city: String
- state: String
- country: String
- zipCode: Integer
Person
- firstName: String
- lastName: String
- address: Address
- gender: Char
- dob: Date
- email:String
- password:String
Admin
+ makeReservation(Passenger):void
+ queryDisplayCancel(void):void
+ login(void):Boolean
+ signOut(void):Boolean
Customer
- passenger:List<Passenger>
- noOfReservation:Integer
+ signUp(Person):Boolean
+ login(String,String):Boolean
+ signOut(void):Boolean
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
Passenger
- passportNo: Integer
- visaType: String
- ticketType:String
- mealType:String
+ displayDetails(void):void
Database
- factory: SessionFactory
+ addReservationToDB(Reservation):List<Reservation>
+ addCustomerToDB(Customer):Boolean
+ checkCustomerInDB(Customer):Boolean
+ getFlightListFromDB(List<Tranportation>):AvailableTra
+ getDBTable(void):SessionFactory
Implement the singleton design pattern
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
Flight
+ returnAvailableSeats(void):Integer
Transportation
- airline:String
- aircraft:String
- vesselNo:String
- sourceAirport:String
- destinationAirport:String
- arrivalDate:DateTime
- departureDate:DateTime
- modelName:String
- noOfSeats:Integer
+ returnAvailableSeats(void):integer
Implement the composite design pattern along with its subclasses
AvailableTransport
- availList:List<Transportation>
+ returnAvailableSeats(void):integer
+ searchTransport(void):void
+ addAvailTransport(Transportation):void
+ removeAvailTransport(Transportation):void
SelectedTransport
- selectedList: List<Transportation>
+ returnAvailableSeats(void):integer
+ selectPreference(void):void
+ addSelectedTransport(Transportation):void
+ removeSelectedTransport(Trabsoirtation):void

Figure 3: Refactored Class diagram: Shows the singleton (green shading) and composite (orange shading) design patterns implemented by means of notes in the class diagram