Task-1

Design pattern : Builder Design Pattern

Requirements of the project:

- A platform for customer's to pay online and book a parking spot online
- A platform for parking enforcers to add, remove and monitor parking spot
- A platform for system administrators to add, remove parking enforcement officer and authorize and change payment status.

Why Builder Pattern?

Here, we create a complex object that is built using simple objects in a step by step approach. Additionally, we are separating the implementation and representation of the object and hiding implementation details of the object from the user.

The primary reason to user builder pattern is that it, separates the implementation of a complex object from its representation so that the same construction process can create different representations. In here, we have different representations and different functionalities for the representations. Builder pattern provides the ease of separating them and provides greater efficiency and control over the implementation process.

What requirement can help?

The separation of representation and implementation is helpful here. As we are separating the objects in the implementation part to many other simpler and smaller object to create and build the complex object, changing one part or section of the implementation doesn't impact the other. The separation of representation and implementation also gives a good help from security perspective of designing.

Task-2

Interface: User
Abstract class: BookingOperations
Class:
RegisterCustomer, Login, Payment, ParkingEnforcementOfficer, SystemAdministrator, Customer

Relations between classes:

Customer **implements** User, BookingOpeartions
Customer **associated** with Payment => unidirectional relationship
ParkingEnforcementOfficer **implements** User, BookingOpeartions
SystemAdministrator **implements** User, BookingOpeartions
SystemAdministrator **associated** Payment => Unidirectional relationship
RegisterUser **implements** User
Login **implements** user
Login **associated** with customer, parkingEnforcementOfficer, and SystemAdministrator => **unidirectional relationship** as Login calls on these three classes to check vice versa doesn't happen
Database class has a bidirectional relationship with login. Login calls database and checks through it to see if user exists or not
RegisterClass has unidirectional relation with Database, as it calls and adds user to the database if the customer doesn't exist.

Implementation:

I will use interfaces User and abstract class BookingOperations. The reason for me to do this is a java doesn't allow multiple inheritance and the interface and abstract class have methods that need to be implemented according to the object with one not colliding or impacting each others properties. This design pattern also provides a good scope for code reusability and compensates the absence of multiple inheritance. In both user and BookingOperations we have that are connected logically and therefore needs to be linked, which they are by using abstraction, interface and associations.

Design solution for the requirements:
Req 4.1 manage parking enforcement => in the SystemAministrator class I implement addParkingofficer(String name) and removeParkingofficer( ) to meet this requirement

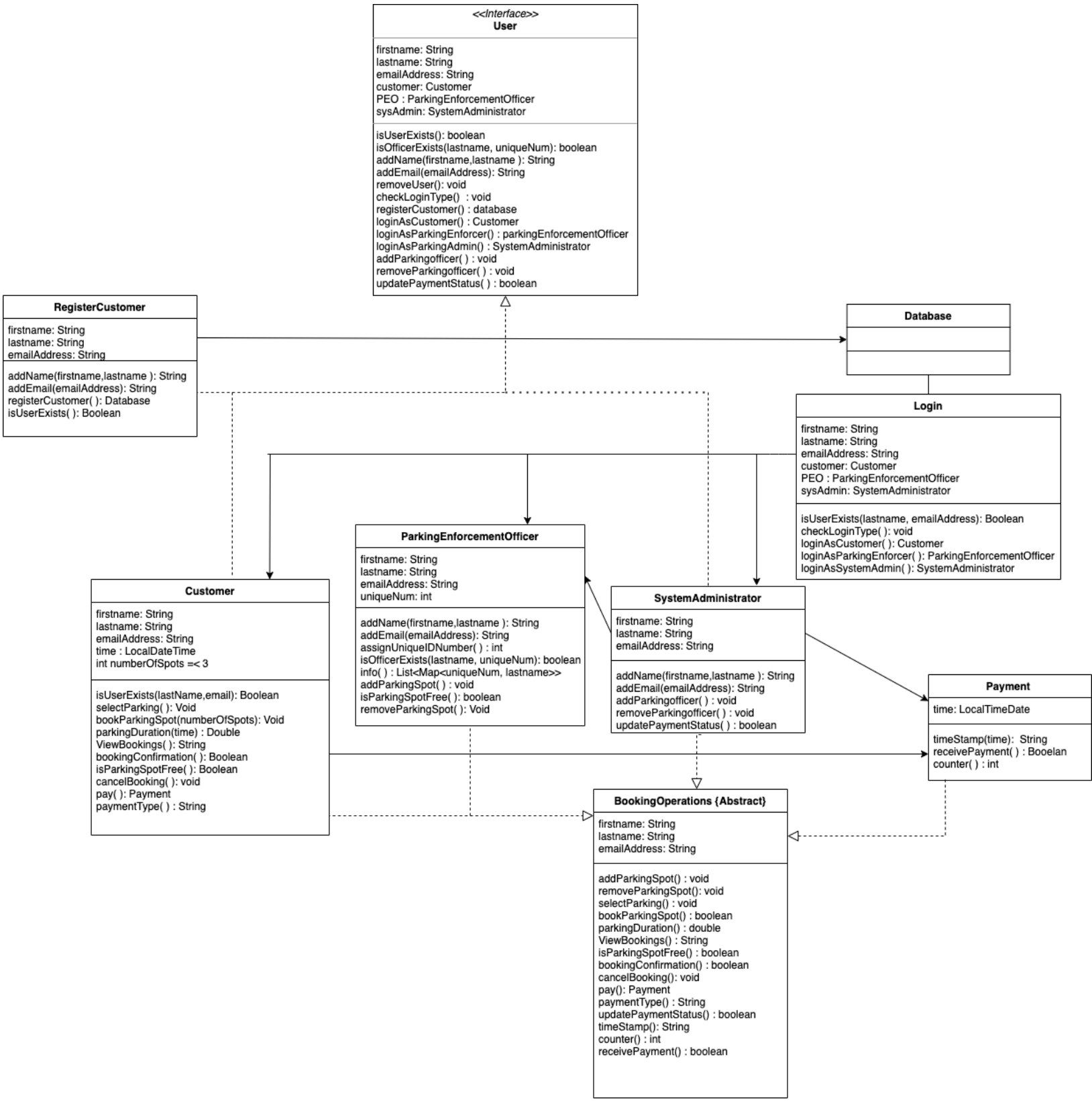
Req 4.2: customer registration => RegisterCustomer class implement the addName(String firstName, String lastName), addEmail(String emailAddress), registerCustomer( ), isUserExists(lastName, emailAddress) to meet the requirements where the registerCustomer method connects with the database to store the customer information

Req 4.3: user login => Login class implements user and is associated with database [bi-directional direction] when we are at login the important part here is to confirm the role of the user, using the checkLoginType( ) method and also use isUserExists for pre existing user. In the checkLoginType we have Methods which connects to the respective classes and takes the user to their respective roles defined in the classes dedicated to them
Ie, if (user == customer) loginAsCustomer( ),
if (user == parkingGuy) loginAsParkingGuy( )
if (user == admin) loginAsAdmin( )

Req 4.4 - 4.7: Book a parking space => the BookingOpeartions interface methods, bookSpot( ) method and parkingDuration( ) method, selectParking( ), bookParkingSpot( ), parkingDuration( ), ViewBookings( ), isParkingSpotFree( ), bookingConfirmation( ), bookingCheck( ), cancelBooking( ), pay( ), paymentType( ). Implemented in the customer class deals with the requirements

Req 4.8: addParkingSpot() and removeParkingSpot() methods in Parking enforcement class deals with this requirements

Req 4.9 change payment status => updatePaymentStatus( ) from the bookingOperations interface deals with the requirements


<<Interface>>
User
firstname: String
lastname: String
emailAddress: String
customer: Customer
PEO : ParkingEnforcementOfficer
sysAdmin: SystemAdministrator
isUserExists(): boolean
isOfficerExists(lastname, uniqueNum): boolean
addName(firstname,lastname): String
addEmail(emailAddress): String
removeUser(): void
checkLoginType() : void
registerCustomer() : database
loginAsCustomer() : Customer
loginAsParkingEnforcer() : parkingEnforcementOfficer
loginAsParkingAdmin() : SystemAdministrator
addParkingofficer( ) : void
removeParkingofficer( ) : void
updatePaymentStatus( ) : boolean
RegisterCustomer
firstname: String
lastname: String
emailAddress: String
addName(firstname,lastname): String
addEmail(emailAddress): String
registerCustomer( ): Database
isUserExists( ): Boolean
Database
Login
firstname: String
lastname: String
emailAddress: String
customer: Customer
PEO : ParkingEnforcementOfficer
sysAdmin: SystemAdministrator
isUserExists(lastname, emailAddress): Boolean
checkLoginType( ): void
loginAsCustomer( ): Customer
loginAsParkingEnforcer( ): ParkingEnforcementOfficer
loginAsSystemAdmin( ): SystemAdministrator
ParkingEnforcementOfficer
firstname: String
lastname: String
emailAddress: String
uniqueNum: int
addName(firstname,lastname): String
addEmail(emailAddress): String
assignUniqueIDNumber( ) : int
isOfficerExists(lastname, uniqueNum): boolean
info( ) : List<Map<uniqueNum, lastname>>
addParkingSpot( ) : void
isParkingSpotFree( ): boolean
removeParkingSpot( ): Void
Customer
firstname: String
lastname: String
emailAddress: String
time : LocalDateTime
int numberOfSpots =< 3
isUserExists(lastName,email): Boolean
selectParking( ): Void
bookParkingSpot(numberOfSpots): Void
parkingDuration(time) : Double
ViewBookings( ): String
bookingConfirmation( ): Boolean
isParkingSpotFree( ): Boolean
cancelBooking( ): void
pay( ): Payment
paymentType( ) : String
SystemAdministrator
firstname: String
lastname: String
emailAddress: String
addName(firstname,lastname): String
addEmail(emailAddress): String
addParkingofficer( ) : void
removeParkingofficer( ) : void
updatePaymentStatus( ) : boolean
Payment
time: LocalTimeDate
timeStamp(time): String
receivePayment( ) : Booelan
counter( ) : int
BookingOperations {Abstract}
firstname: String
lastname: String
emailAddress: String
addParkingSpot() : void
removeParkingSpot(): void
selectParking() : void
bookParkingSpot() : boolean
parkingDuration() : double
ViewBookings() : String
isParkingSpotFree() : boolean
bookingConfirmation() : boolean
cancelBooking(): void
pay(): Payment
paymentType() : String
updatePaymentStatus() : boolean
timeStamp(): String
counter() : int
receivePayment() : boolean