Justification for choosing these six designs patterns

1. Proxy

This design pattern was used to provide an intermediary between the user object and the facade object in order to control the access to the facade.

2. Facade

The facade was used to represent the space booking and management subsystems to hide their complexities from the client. This makes the subsystems easier to use and navigate.

3. Strategy

This design pattern helped the client switch between different payment modes which required different logic to execute processPayment().

4. State

State was used to change the behaviour of scanSpace() of the parkingSpace object depending on whether its state was disabled or enabled.

5. Singleton

This design pattern was used to ensure that if a client or manager logs out and logs back in under the same or different account, a new client or manager object would not be created again and instead, return the previously created object but update its fields if need be.

6. Iterator

Iterator was used to iterate through the map of parkingSpace objects in parkingLot.

Discussion about how each requirement can be achieved

Req1: Any client should be able to register as a user. Allows four types of clients to be registered. If a client registers as a student, a faculty member or a non-faculty staff, her/his registration requires a further validation from the management teams.

This requirement is achieved by a login/register page that users can use to register an account. The User class has a register() method to do this. Furthermore, one of the fields they must register with is “Client Type”. If the selected client type is student, faculty member, or non-faculty staff, they must also provide “ExtraId” for managers to retrieve and verify. Managers have a displayPendingValidations() method to show all clients that still require validation and a validateClient() method to do this.

Req2: Super manager (only one) can auto generate manager accounts.

To prevent creation of more super manager accounts, we did not create a registerSuperManager() method. Instead, the SuperManager object is created if needed by reading the csv file. The SuperManager object also has a autoGenerateManager() account to generate new manager accounts.

Req3: Any registered client can book a valid parking space after login to the system, the parking rates for different types of users are different, i.e., 5$, 8$, 10$, and 15$ for students, faculty members, non-faculty staffs, and visitors per hour respectively.

The client object confirms booking with clientType field (among other fields) by forwarding the request to the Proxy, which then forwards it to the Facade which then forwards it to BookingManager. BookingManager then asks SpaceManagers to check for space availability and also asks PaymentManager to calculate cost depending on client type and handle payment.

Req4: Booking a parking space requires the cost of an hour (of the type of a client) as the deposit, which will not be refunded if a no-show happens in the first 1 hour of the booked parking period. Otherwise, the deposit will be deducted when checking out.

Cost depending on client type + deposit is calculated by PaymentManager object. PaymentManager has a refundDeposit() method for when a client checks out. It also has a refundWithoutDeposit() method if a no-show happens. Deposit is kept and the client is refunded the rest.

Req5: Suppose each parking space has a sensor to detect if a car is using the parking space or not. In addition, the sensor can also scan the basic info of cars, and further send the essential information to the system.

ParkingSpace object has a scanSpace() method that managers can call to return the licensePlate of a checked in client, return “empty” if space is enabled but not checked in, and return “disabled” if the space is disabled.

Req6: Managers of the system can add, enable, or disable a parking lot, a parking lot contains 100 parking spaces. Managers can also enable or disable a parking space due to maintenance issues.

Managers have a addParkingLot(), toggleSpace(), toggleLot() method that gets forwarded down the same proxy and facade chain until it reaches SpaceManager since this object stores the parking lots. Each ParkingLot object contains a map of parking spaces that have a limit of 100. Spaces are only instantiated if needed (booking or disabling) to save memory by not creating 100 spaces at program launch.

Req7: Each parking space will have a unique identification number and other details including its location and its parking lot, which will help with the navigation for clients.

ParkingSpace objects have spaceId as a field and ParkingLot objects have lotId as a field. This information can be displayed when the client heads over to the “My Bookings” tab in order to check in or find the space location.

Req8: To book a parking space, clients need to provide a valid licence plate number. A client can edit or cancel her/his bookings before the starting time of a booking.

Clients have editBooking() and cancelBooking() methods that get forwarded down the same chain to BookingManager where the logic is executed. Clients must also provide their license plate when using the confirmBooking() method.

Req9: Clients can extend a parking time before the expiration.

Clients have extendBooking() method that gets forwarded down the same chain to BookingManager where the logic is executed.

Req10: Clients can pay their parking fee via different payment options, i.e., debit cards, credit cards, mobile payments, etc

Clients can toggle to pay with debit/credit or mobile pay with a button and PaymentManager’s processPayment() method is adjusted based on the chosen payment method.

Justification for component decomposition and interactions

Clients, SuperManager, and Managers use the User object when the program first runs. User object then requires the Proxy object in order to register or login. Proxy requires AccountRegistry to check if login information matches an entry in the registry and to update the registry for new registrations. Upon successful login, a Client, SuperManager or Manager object is created by Proxy. Thus Proxy requires both Client and Manager. SuperManager requires Manager since SuperManager extends Manager. User, Client, Manage, and SuperManager are in the Login/Register subsystem because these are the initial objects created when a user first interacts with the system.

Proxy requires Facade because Proxy forwards all requests from users to the Facade. Facade requires the CheckInManager, BookingManager, and SpaceManager because Facade forwards all requests from users to them to execute the logic. CheckInManager requires BookingManager because CheckInManagers needs to see the active bookings in BookingManager in order to check in to the right bookings. Both Facade and BookingManager require SpaceManager because they both need access to the map of parking lots. Booking Manager also trivially requires Booking. Proxy, Facade, AccountRegistry, BookingManager, CheckInManager, and Booking are in the ParkingSpaceBooking subsystem because all these objects mostly focus on creating bookings for the client and managing said bookings.

Booking Manager requires PaymentManager to process payment and refunds for bookings. PaymentManager requires MobilePay and DebitOrCredit to execute the correct logic depending on how the client wants to pay. PaymentManager, MobilePay, and DebitOrCredit are in the Payment subsystem because the 3 of them are solely responsible for processing payments and refunds.

SpaceManager and AccountRegistry both require CSVManager in order to read the csv files. SpaceManager also requires ParkingLot since SpaceManager manages a map of ParkingLot objects. ParkingLot requires ParkingLotIterator and ParkingSpace since it manages a map of ParkingSpace objects. ParkingSpace requires Enabled and Disabled to adjust its state. SpaceManager, ParkingLot, ParkingLotIterator, ParkingSpaces, Enabled, and Disabled are in the ParkingSpaces subsystem because this is where all the data of parking spaces and lots are stored and manipulated.

CSVManager is in its own subsystem of CSVReading because it is solely responsible for reading and updating the csv files and passing the information along to other objects that need it.