Low Level Design

for

Hotel Management System

Table of Contents

1. Components

The major components for our Hotel Management Systems are :

1. Login into the HMS
2. Search
3. Reservations by payment
4. Cancel reservation
5. Database
6. Update employee or hotel info
7. Revenue Maintenance

1. Login into the HMS

This component of our system gives the user/employee with a valid user ID and ability to set the password. This gives the permissions and authorizes the data such as updating employee information, hotel features, maintaining the revenue etc., according to the hierarchy. Everytime when the user gives credentials it interacts with the database to check the validity and permissions for those credentials.

2. Search

This component of our system allows the user to search for the details like rooms, amenities etc., User can reserve the rooms by searching in the web portal or mobile platform. It also gives an option to search room availability by date, time, features or price. It interacts with the database to get the details of the rooms and their availability.

3. Reservations by Payment

This component of our system allows the user to pay online for reserving the rooms. It interacts with the database and the online banking to validate and process the payment. It requires the details of the user essential for processing. Then it stores the details of the user and updates the availability of rooms in the database.

4. Cancel Reservation

The cancel reservation component performs the cancellation of a room online. When it is requested by the user, it checks for the user details and any payments due as per rules and then cancels the reservation. It updates the room availability details in the database.

5. Database

This component of our systems takes active participation and included for many activities. It stores all the information of the customer details, room availability, hotel features, revenue and employee information. The components can retrieve and store the information from/into the database component.

6. Update employee or Hotel info

This component takes the input from the Login component, i.e., the login authorization and then allows the user to make changes such as, add or delete employees, add hotel features, pictures, rooms, change prices, etc. The changes made will be updates into the database.

7. Revenue Maintenance

The revenue maintenance component also takes the login authorization as input to allow the Owner to review the revenue details of the hotel such as tax, income, capital, etc. The component retrieves the information from the database and store any changes back into the database.

2. Cohesion

Cohesion refers to the degree to which the elements inside a module belong together. In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's methods and data themselves.

In our context of component level design cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

These are the different types of cohesions in our components.

1. Search

Coincidental Cohesion matches the search module of HMS. Coincidental cohesion is the lowest level of cohesion possible; essentially it is no cohesion. The elements in the component have no discernable relationship to each other. The activities are not related through data or control flow, and may not even be in the same category. For e.g.

- Search room
- Refresh page
- Look for pictures
- Know facilities offered
- Filtering options (search by price, date, etc)

2. Reserve by Payment

For this component **Logical Cohesion** is the best fit. With logical cohesion, the elements in the module have some general link to each other, although it may be a weak one. For e.g.

- Pay by credit card
- Pay by debit card
- Pay by PayPal
- Pay by electronic check

The activities are related in that they are all ways to pay, but at any given time you must choose one of the methods to use.

3. Cancel Reservation

Procedural Cohesion exists for this component. Procedural cohesion is present when the elements of the component, although possibly unrelated, are linked by control flow. They may be related to activities in other components. For e.g.,

- Cancel Room/Reservation
- Update availability of rooms in the portal

These activities are defined to occur in a specific order, but not necessarily at any specific time. The activities have little relation to each other except for the required order.

4. Database

We can observe **Communication Cohesion** in this component. Communication cohesion is present in a module when the elements all use the same input or output data. For e.g.,

- Find user details
- Find employee details
- Find hotel features
- Retrieve revenue details

The elements are related because they all use the same data structure, database. Components of this type are generally maintainable.

5. Update Employee or Hotel Information

Sequential Cohesion exists in this component. With sequential cohesion, activities are based on data flow. The output from one activity serves as the input to the next. For e.g.,

- Login
- Authorization
- Add/delete employee or update hotel features

There is a sequential flow of data between the elements of the component, the manager logins, if it authorized, then only he can make changes in hotel features or employee details.

6. Revenue Maintenance

We have **Functional Cohesion** for this component. The activities in a component with functional cohesion work towards one and only one problem-related task. For e.g.,

- Review employee salaries
- Review tax details
- Look for income from customers
- Check for the capital invested in hotel maintenance

All these activities work toward one and only goal of maintaining revenue of the hotel which is done by the owner.

3. Coupling

Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are the strength of the relationships between modules.

Coupling is a qualitative measure of the degree to which components are connected to one another. For low coupling, connections should be narrow, direct, local, obvious, and flexible.

1) There exists a **Content and Common Coupling** between a Database component and each of the other component, namely Login, Search, Reserve by payment, Update employee and hotel info, Cancel Reservation and Revenue Maintenance.

Content coupling occurs when one module relies on or modifies the internal workings of another module. This has also been called **Pathological coupling**. Clearly, it results in little or no independence between the modules. Common coupling occurs when two or more modules share the same global data structure or resource. In our system, all the components share the same global data structure - Database. If the resource changes, all modules using it must change, i.e., whatever changes are made in updating employee details, revenue details, reservation cancellation, all these changes are reflected in the Database.

2) We have **Control Coupling** between the components Login and "Update Employee and Hotel Info" or "Revenue Maintenance". Control coupling occurs when one module controls the operation of another, passing it information on what to do. When the manager or owners wants to update the employee or hotel features or review revenue details, they should first login. If the login is authorized, then only they can proceed further to perform those activities. Thus, one module controls the operation of another.

3) There exists **Data Coupling** between Database and "Cancel Reservation" components. Data coupling passes only the data the other module actually needs. For e.g. Module A and Module B is data coupled if A calls B and B returns to A. All information passed between them are through parameter passing. Each parameter is a data element. In our system, to cancel a reservation, the parameters needed from the database are Room number and Payment information. Thus the component "Cancel Reservation" calls the "Database" and the Database returns these two parameters.

4. Design Patterns:

Design patterns provide solutions to common software design problems which had occurred in the previous designs. Design patterns complement the high-level architectural styles by presenting models that can be used to guide the development of an effective design in specific situations. Design patterns are a powerful tool for a software developer. Most often software developers do not understand the concept in a design pattern and just copy the classes, methods and properties. But it is important that developers understand the concept involved behind the usage of design pattern. A design pattern is not a finished solution but a guide that can be moulded and acts as a way to start a code for a developer.

Design pattern comprises of the following:

- In a design pattern solutions are expressed in terms of classes of objects and interfaces.
- A design pattern names, abstracts, and identifies the key aspects of a high quality design structure that make it useful for creating reusable object-oriented designs.

Design Patterns template can be formed by usage of the following:

1) Name
2) Problem
3) Solution
4) Consequences and trade-of of application
5) Implementation: An architecture using a design class diagram.

Types of Design Patterns:
There are currently three design patterns available. They are:
1) Creational Design Pattern: These design patterns are about the class instantiation. Creational patterns provide ways to instantiate single objects or groups of related objects. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.
2) Structural Design Pattern: These design patterns are about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.
3) Behavioural Design Pattern: These design patterns are all about Class objects and how they communicate. Behavioral patterns identify communication patterns between objects and increase flexibility in implementing communication.

We think the following 3 patterns would be suitable to the current design of the **Hotel Management System**. They are:

1) Façade Pattern
2) Strategy Pattern
3) Command Pattern .

<u>1) Facade Pattern:</u>

Facade design pattern simplifies the interface of a complex system. It is a collection of all the classes which make up the subsystems of the complex system. This design pattern. A Facade design pattern eases the user from the complex details of the system and provides them with a simplified view, which is easy to use. This design pattern ensures there's no relation between the code that uses the system and the details of the subsystems, making it easier for the developers to modify the system in the later stages Facade pattern is suitable when there are working with a large number of classes, or with classes that require the use of multiple methods. Facade pattern is a structural pattern as it defines a manner for creating relationships between classes.

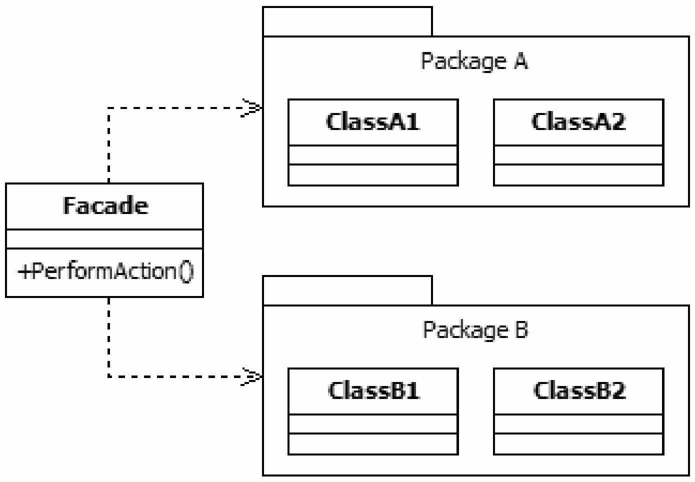

Fig 4.1: Implementation of Facade Pattern

Implementation Details:
The UML diagram for the Facade pattern can be explained by the following implementation details.

- Facade: This class contains the functions that are used by users and are visible, and the complex subsystems are hidden.
- Packages (A/B): All the complex classes used in Facade design need not be only under one package. They can be under multiple packages.
- Classes (A/B,1/2): These classes contain the functionality that is being presented via the facade.

Name	Façade Design Pattern
Intent	To simplify the existing system to a user.

Problem	Need to use only a subset of a complex system in facade pattern.
Solution	The façade pattern presents a new interface for the client of the existing system to use.
Consequences	Façade pattern simplifies the use of required subsystem. But certain functionality needs to be added based on the system.
Implementation	Defines new classes that has the required interface and uses the existing system.

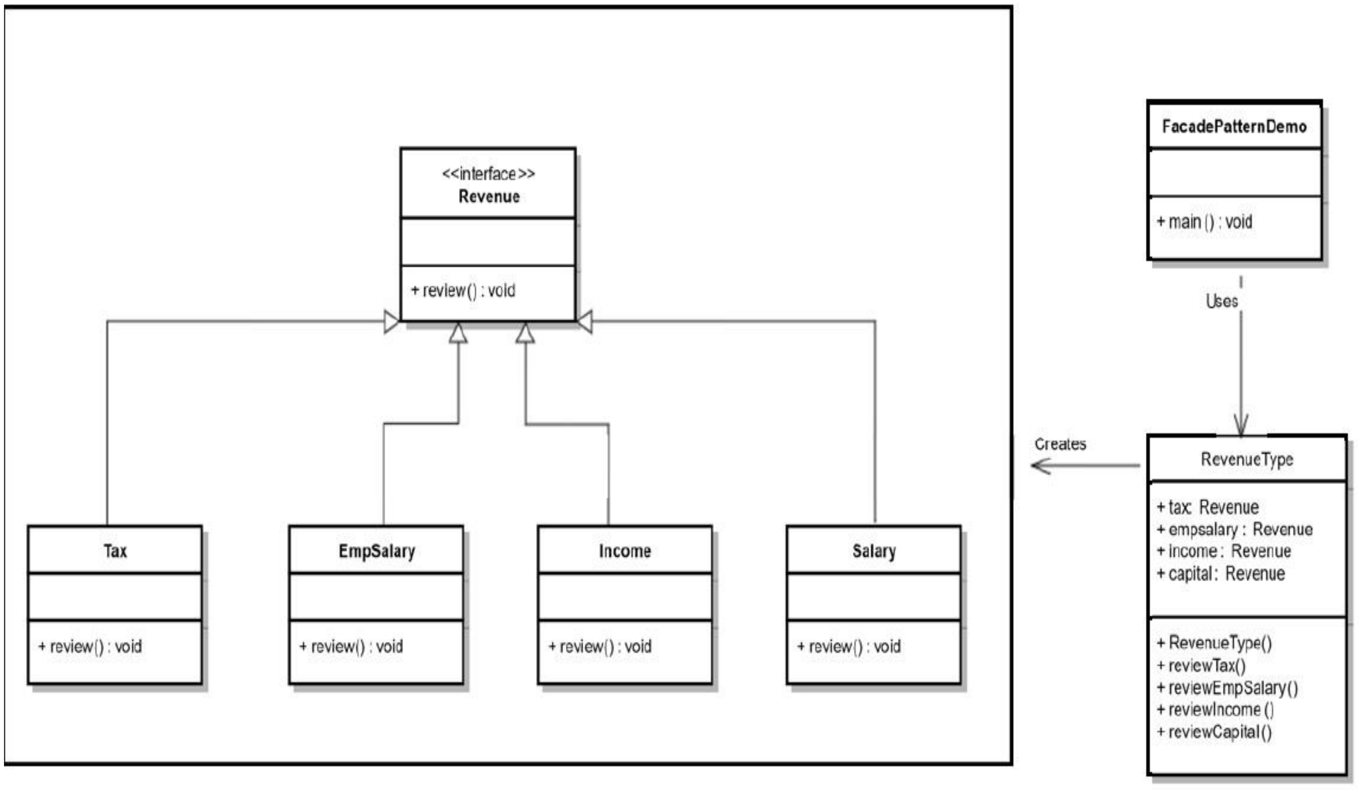

Fig 4.2: UML for Facade Pattern for revenues

<u>2) Strategy Pattern:</u>

The Strategy Pattern is a type of Behavioral pattern which allows the algorithm vary independently from clients that use it. The Strategy Pattern Context class has multiple control strategies provided by the concrete strategy classes, or by the abstract strategy classes. The pattern enables an algorithm's behavior to be selected at runtime depending on the conditions of the system. It reduces coupling by having the client class be coupled only to the context class.

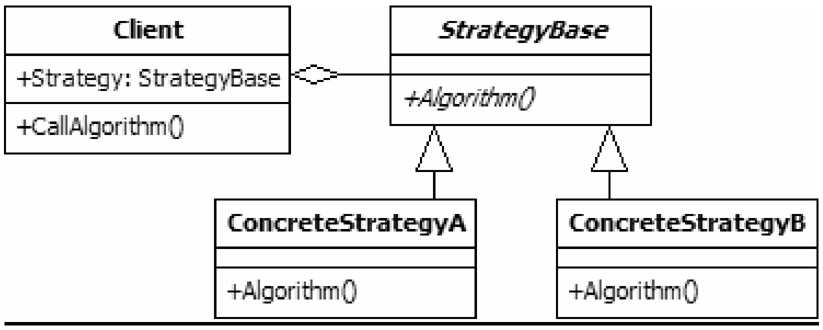

Fig 4.3: Implementation of Strategy Pattern

Implementation Details:

The UML diagram for the Strategy pattern can be explained by the following implementation details.

- Client: The class holds a property of one of the strategy classes. This property will be set at run-time according to the algorithm that is required.
- StrategyBase: This is a abstract class that provides algorithms to other classes. This class can also be implemented as an interface if it provides no real functionality for its subclasses.
- ConcreteStrategy A/B: The concrete strategy classes is inherited from the StrategyBase class. Each provides a different algorithm that may be used by the client.

Name	Strategy Pattern
Intent	It allows you to use different business rules depending upon the context in which they occur
Problem	The selection of an algorithm that needs to be applied depends upon the client making the request or the data being acted upon. If there is a rule in place that does not change, there is no need for a strategy pattern.

Solution	It separates the Selection of algorithm for the implementation of the algorithm & allows for the selection to be made based upon context.
Consequences	The strategy pattern defines a family of algorithms. Switches and/or conditions can be eliminated.
Implementation	The context class that uses the algorithm will contain the strategy class which has a abstract method specifying how to call a particular method.

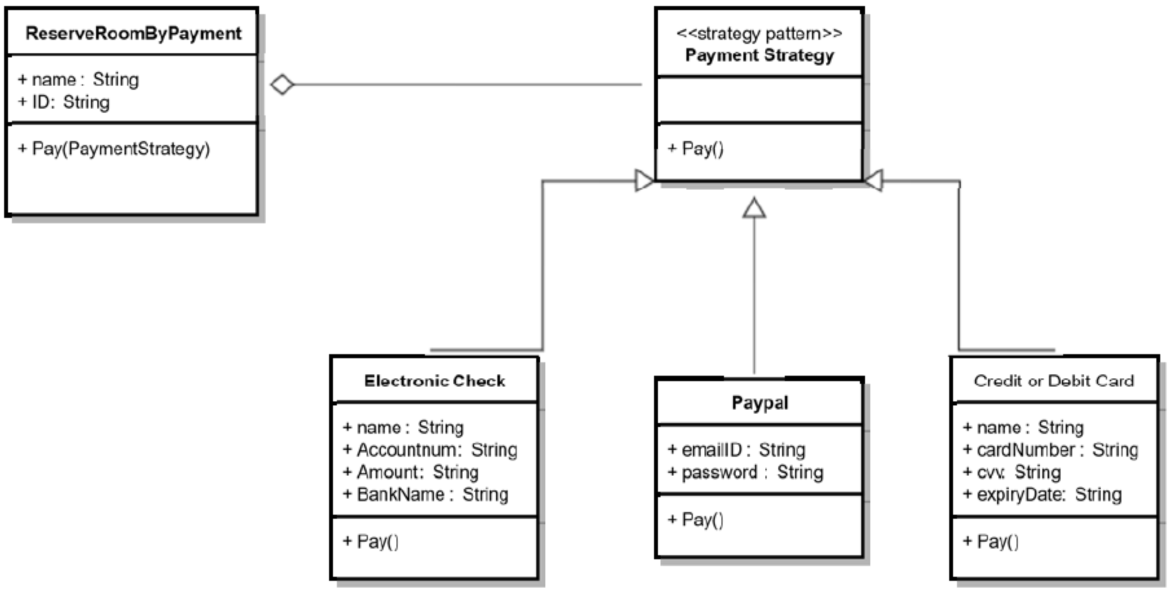

Fig 4.4: UML for Strategy Pattern to reserve by payment

3) Command Pattern:

Command pattern is used to encapsulate all the information that is required to invoke an action or trigger an event in the future time. Command pattern is used with the object oriented programming environment. The information that is encapsulated by the pattern includes a method name, object that owns the method and values for the method parameters. Four terms are always associated with the command pattern: command, receiver, invoker and client. This pattern allows us to achieve complete decoupling between the sender and the receiver. A sender is an object that invokes an operation and receiver is an object that receives request to execute an operation. Decoupling helps to prevent the sender from knowing about the receivers interface. The term request here refers to the command that is to be executed. Command pattern provides us flexibility as well as extensibility.

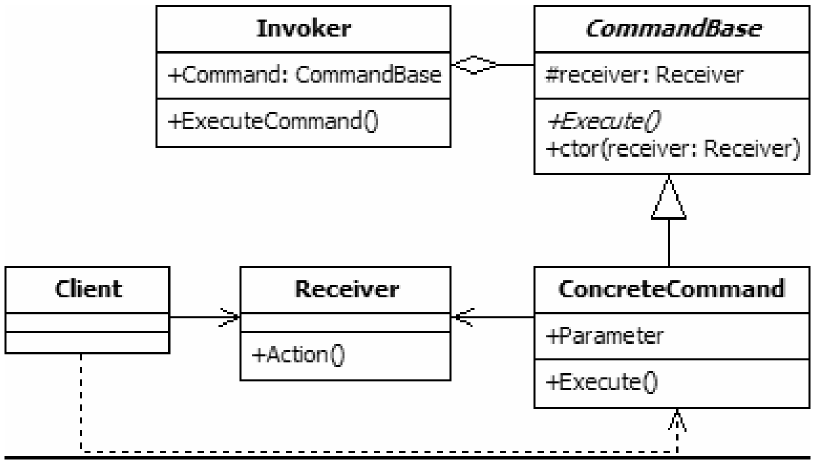

Fig 4.5: Implementation of Strategy Pattern

Implementation Details:
The UML diagram for the Command pattern can be explained by the following implementation details.

- Client: This class creates the command objects and links them to receivers.
- Receiver: Receiver objects contain the methods that are executed when one or more commands are invoked.
- CommandBase: This abstract class that defines a protected field and holds the Receiver that is linked to the command.
- ConcreteCommand: Concrete command classes are subclasses of CommandBase. They contain all of the information that is required to implement Receiver object.
- Invoker: The Invoker object initiates the execution of commands.

Name	Command Design Pattern
Intent	To encapsulate a request in an object and allow parameterization of clients with different requests.
Problem	To issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Solution	The command pattern establishes a command interface to respond to the requests of the sender.
Consequences	Use of the command pattern will allow the user to know the internal functioning of a system with the issues and execution of the commands.
Implementation	The client asks for a command to be executed and invoker takes the command and encapsulates, while the concrete command in charge of the request command sends its result to the receiver.

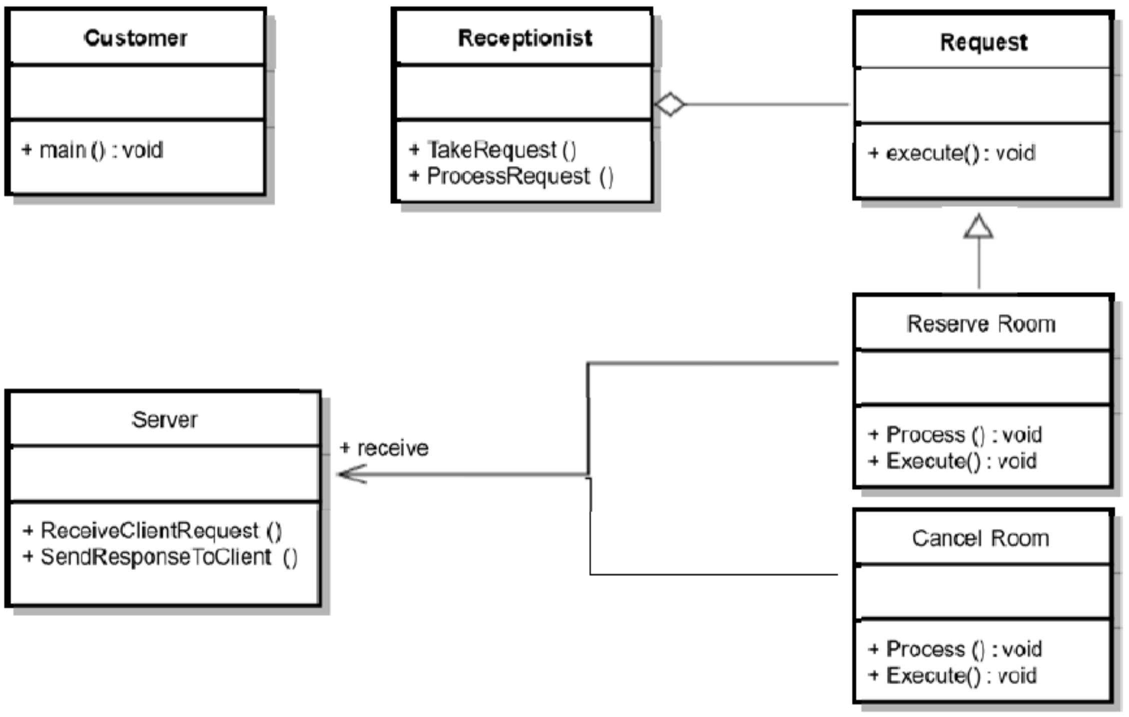

Fig 4.6: UML for Command Pattern for room reservation