Hotel Management System

20200808058 – Tahir Emre Semiz

Project Purpose And Scope:

The main purpose of this project is to create a hotel management system that is more efficient, understandable, flexible and user-friendly by using 4 different design patterns. To design and implement a Hotel Management System that prioritizes mediocrity and addresses the complex challenges of managing hotel rooms, reservations and customer interactions.

Room Management: A system where defining and arranging different room types and adding new room types hassle-free

Reservation System: Designing a central control mechanism using the Singleton Model

Customer Notifications: Customers are informed in real time through the reservation system via Observer Pattern.

Service Enhancement: Dynamically increasing room functions by applying the Decorator Model allows the addition of extra services.

Design Patterns: Detailed Explanations:

Each of the design patterns in this project serves a different purpose and role. These patterns are to improve the code quality of the system environment. Here is a list of these design patterns and the purposes and roles of each:

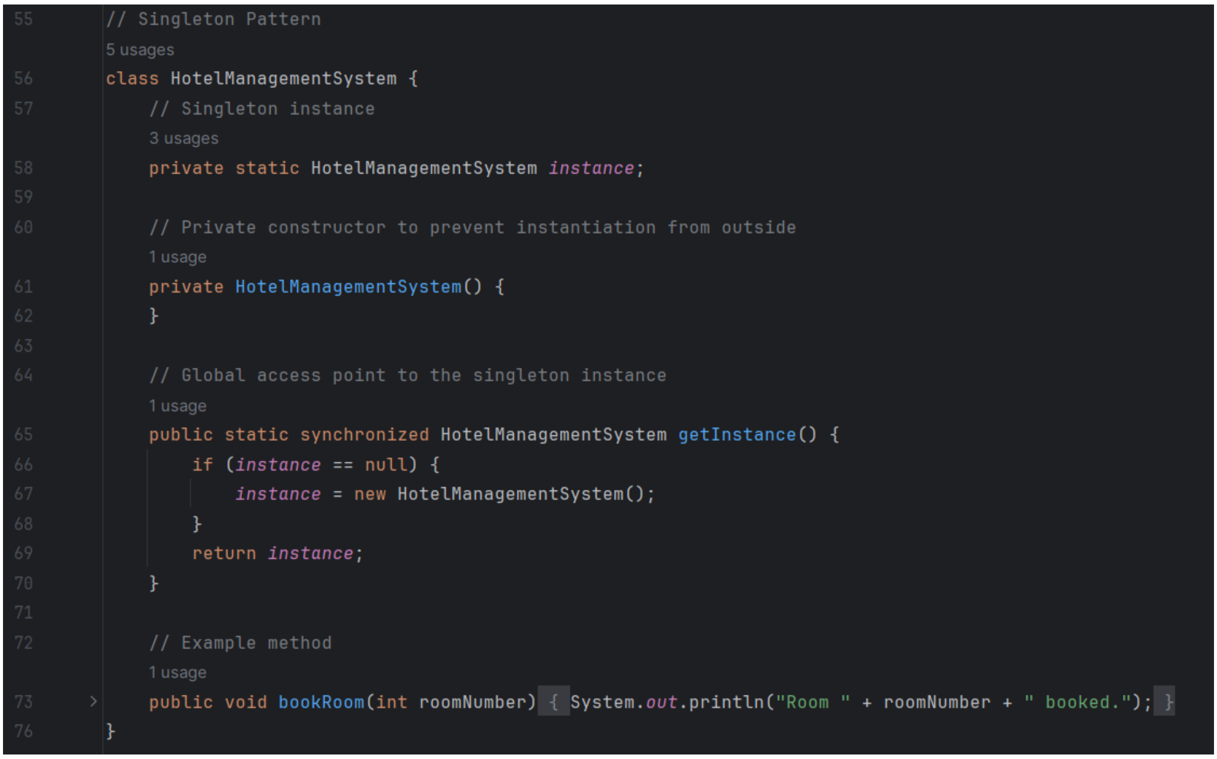
1. Singleton Pattern:

Single Global Instance: Provides a central control center for hotel operations by providing a single instance of the Hotel Management System. avoids conflicts in managing hotel-wide functionalities.

Global Access Point: Provides an entry point to manage and coordinate critical hotel functions (e.g. creating a new room).

Consistent Status: Ensures all components reference the same instance when interacting with the hotel management system, ensuring consistent state across the application

The code implementation is as follows (under improvement):



```
// Singleton Pattern
5 usages
class HotelManagementSystem {
    // Singleton instance
    3 usages
    private static HotelManagementSystem instance;

    // Private constructor to prevent instantiation from outside
    1 usage
    private HotelManagementSystem() {
    }

    // Global access point to the singleton instance
    1 usage
    public static synchronized HotelManagementSystem getInstance() {
        if (instance == null) {
            instance = new HotelManagementSystem();
        }
        return instance;
    }

    // Example method
    1 usage
    public void bookRoom(int roomNumber) { System.out.println("Room " + roomNumber + " booked."); }
}
```

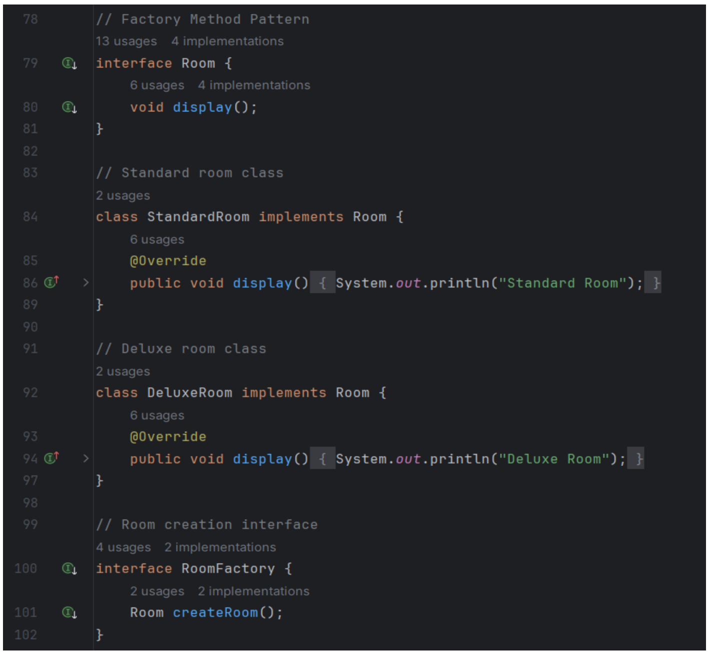
2. Factory Method Pattern:

Abstract Creation Process: For abstracting the creation process of room instances.

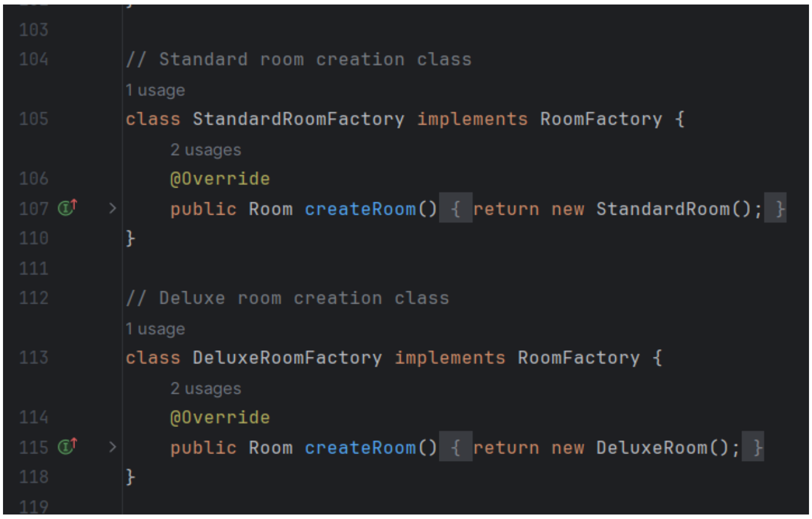
Flexible Creation: Facilitates the addition of new room types and ensures a flexible approach to creating different room categories.

Creation Logic: It keeps the creation logic of each room type encapsulated within the corresponding factory class.

The code implementation is as follows (under improvement):



```
// Factory Method Pattern
13 usages   4 implementations
interface Room {
    6 usages   4 implementations
    void display();
}

// Standard room class
2 usages
class StandardRoom implements Room {
    6 usages
    @Override
    public void display() { System.out.println("Standard Room"); }
}

// Deluxe room class
2 usages
class DeluxeRoom implements Room {
    6 usages
    @Override
    public void display() { System.out.println("Deluxe Room"); }
}

// Room creation interface
4 usages   2 implementations
interface RoomFactory {
    2 usages   2 implementations
    Room createRoom();
}
```



```java
// Standard room creation class
class StandardRoomFactory implements RoomFactory {
    @Override
    public Room createRoom() { return new StandardRoom(); }
}

// Deluxe room creation class
class DeluxeRoomFactory implements RoomFactory {
    @Override
    public Room createRoom() { return new DeluxeRoom(); }
}
```

3. Observer Pattern:

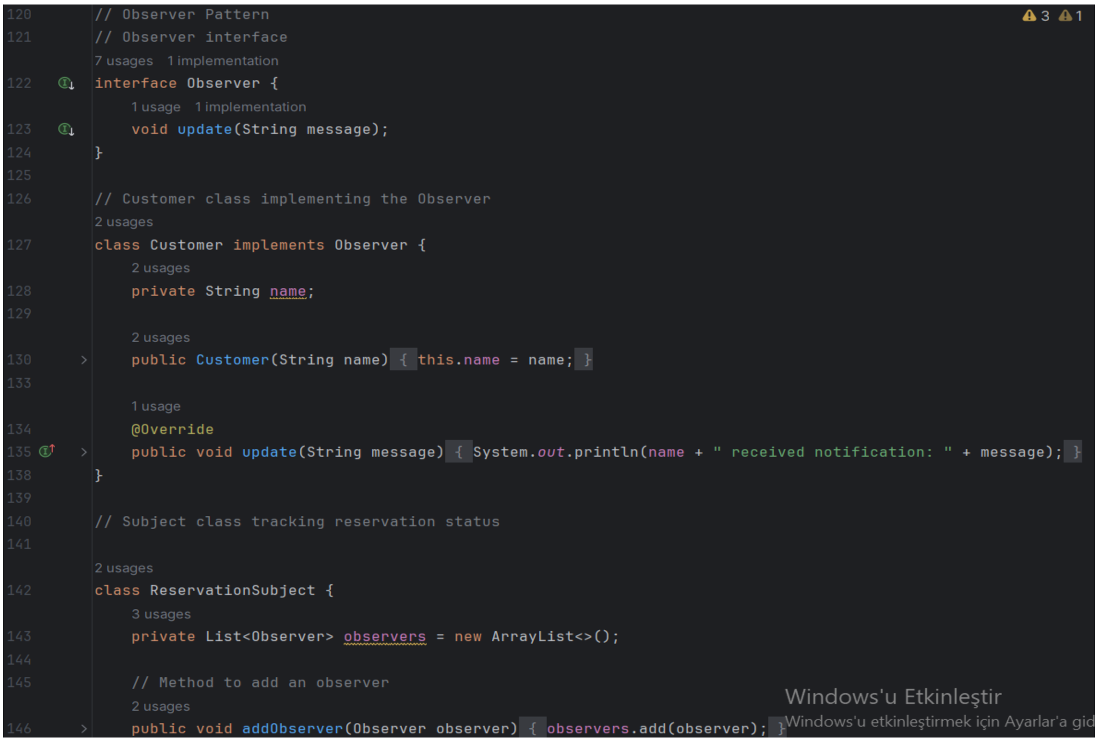
Dynamic dependency: Creating a dynamic one-to-many dependency between the reservation system and customers.

Live Status change tracking: Enables customers to receive real-time updates on reservation status changes.

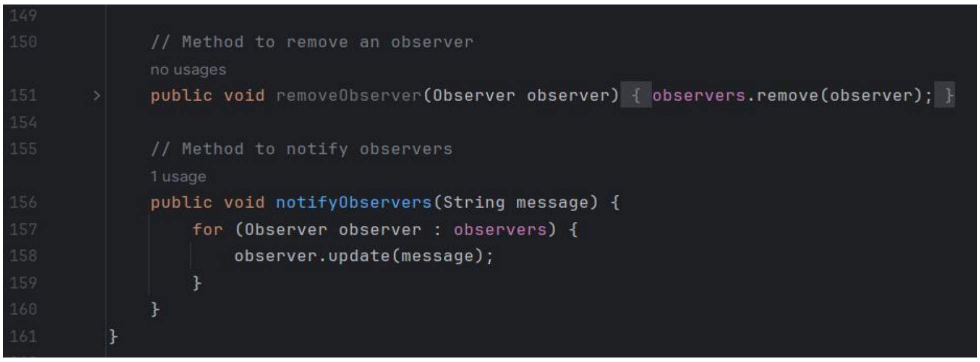
Flexible Subscription: It allows customers to dynamically subscribe or unsubscribe from notifications based on their preferences.

Useful architecture: Reservation status changes do not require changes elsewhere (client code).

The code implementation is as follows (under improvement):



```java
// Observer Pattern
// Observer interface
interface Observer {
    void update(String message);
}

// Customer class implementing the Observer
class Customer implements Observer {
    private String name;

    public Customer(String name) { this.name = name; }

    @Override
    public void update(String message) { System.out.println(name + " received notification: " + message); }
}

// Subject class tracking reservation status

class ReservationSubject {
    private List<Observer> observers = new ArrayList<>();

    // Method to add an observer
    public void addObserver(Observer observer) { observers.add(observer); }
```



```
    // Method to remove an observer
    no usages
    public void removeObserver(Observer observer) { observers.remove(observer); }

    // Method to notify observers
    1 usage
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

4. Decorator Pattern:

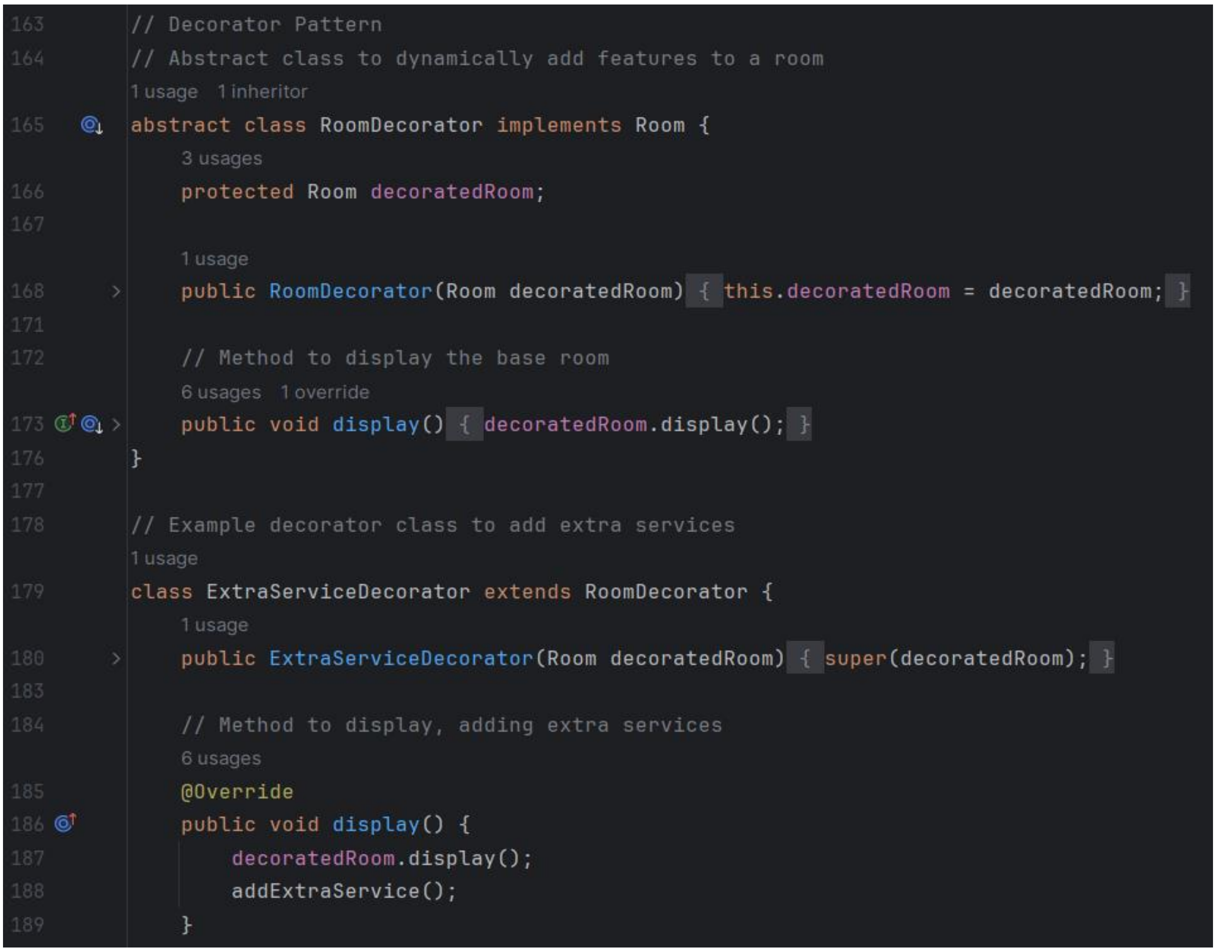
Dynamic Extension: Allows extra services to be dynamically added to room objects via decorators.

Flexible Development: Flexibility by allowing the system to enhance core room functions with extra services at runtime

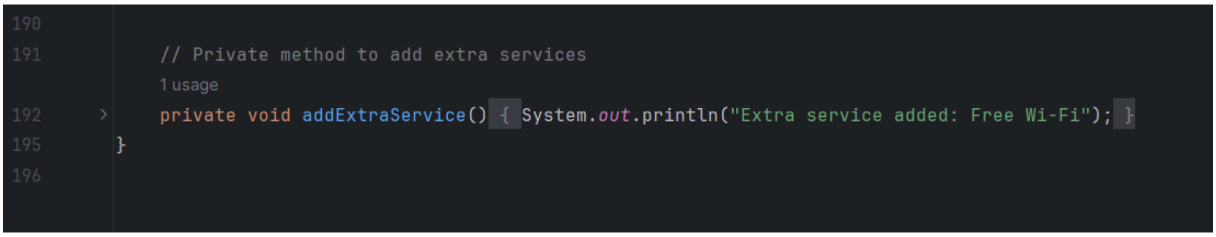
Service Increase: It dynamically enhances basic room functions with extra services by appealing to different customer preferences.

Code Maintainability: It increases the maintainability of the code by ensuring that the addition of new services does not require changes to existing room classes.

The code implementation is as follows (under improvement):



```
// Decorator Pattern
// Abstract class to dynamically add features to a room
1 usage   1 inheritor
abstract class RoomDecorator implements Room {
    3 usages
    protected Room decoratedRoom;

    1 usage
    public RoomDecorator(Room decoratedRoom) { this.decoratedRoom = decoratedRoom; }

    // Method to display the base room
    6 usages   1 override
    public void display() { decoratedRoom.display(); }
}

// Example decorator class to add extra services
1 usage
class ExtraServiceDecorator extends RoomDecorator {
    1 usage
    public ExtraServiceDecorator(Room decoratedRoom) { super(decoratedRoom); }

    // Method to display, adding extra services
    6 usages
    @Override
    public void display() {
        decoratedRoom.display();
        addExtraService();
    }
```



```
    // Private method to add extra services
    private void addExtraService() { System.out.println("Extra service added: Free Wi-Fi"); }
}
```

Conclusion:

In brief, selected design patterns were adopted to meet specific challenges in the hotel management system, including centralized control, scalability, real-time updates, and dynamic service augmentation. Each pattern aids in building a flexible, modular, and easily maintainable system that can accommodate the changing demands of the hospitality sector.

Timeline:

- **27 November 2023:** The hotel management system idea was put forward and work on its architecture began.

- **28 November 2023:** Coding implementation including 4 design patterns was completed. uml diagram was made.

- **1 December, 2023**: Most parts of project propasal have been completed.

- **10 December, 2023:** A total of 7 design pattern elements will be implemented by adding the builder pattern, command pattern and state pattern.

- **11 December, 2023:** New functions containing details will be added to the hotel system.

- **12 December, 2023:** Testing phases and bugs will be fixed. UML diagram will be completed.

- **14 December 2023:** Documentation and presentation will be prepared.