Oop-Final assignment

Ben Shalev- 204818538

1) General explanation:
    a. The system is software system for an online travel agency, similar to platforms like Booking or Airbnb. This system is manage hotel reservations based on various search filters and availability. The system allows hotel owners to control and manage their hotels . In addition, the system allows the customer to search, place orders and manage his orders. In developing the program, I made sure to use design patterns from the object-oriented world, which I will expand on later.

    b. How does it work?
    First we will have a look on the owner side and then at the customer side

    Hotels owner's:

    The program allows the hotel owner to perform the following actions:
    Adding a hotel to the system according to the following characteristics - country, city, name, whether there is parking, whether there is Wi-Fi, whether meals are served, whether there is a bar, whether there is a spa, whether animals are allowed, whether there is a gym, the number of stars and what is the distance from the city center. The hotel is automatically added to the list of hotels of the specific hotel owner and in addition to the hotel database which is actually a hush map of a hush map according to country and city keys and all this in order to enable a hotel search in an efficient way when the database is particularly large.
    Adding rooms to hotel-
    After adding hotel the owner should add rooms to the hotel. He make it by room types with unique elements. The owner should set the the number of rooms from each type.
    getHotel- Allow the owner to get the hotel by name
    ListHotels-Allow the owner to get all his hotels.
    deleteHotel-Allow the owner to delete hotel from the system
    getHotel over/under capacity- Allow the hotel to set amount of percentage for utilization of the hotel's contents for a certain period of time to identify unprofitable hotels and outstanding hotels.
    Get hotel rank-The average rank each hotel get from the customers
    Get next reservation-Allow the owner to see the next reservation by hotel name
    Get reservation by date-Same as previous function just by date
    Send message – Send message to all customers has reservations to specific hotel with specific dates.

Customer:
The program allows the customers to perform the following actions:
Make reservation- the customer enter the hotel details, the dates, the room type and specific adders, this method update the price according the adders. The system automatically reserves a specific room of the same type for the customer, and puts the order on the hotel owner's mailing list.
Cancel reservation- Allow the customer to cancel the reservation and free the room booked.
Get reservation- Show the customer all his reservations.
Get next reservation-Show the customer his upcoming reservation.
Rate hotel-Allow the customer rate hotel.
Update-Allow the customer get messages about his reservation
Search-The search is a multi search. The search allows the customer to choose search filters such as the number of stars, whether there is a pool or a gym. Several search filters can be combined, entering the city and state are mandatory fields to improve search efficiency. In addition, the customer can choose the form of sorting (cheap to cheap or vice versa, according to customer rating or distance from the city center). When the customer receives the search results, he can choose whether to make a hotel reservation.

c. **Things taken into account when developing the software:**

i. Adaptability and Portability:

- Designing for future-proof code that can accommodate changes without system-wide disruptions.
- Utilizing exception handling for error management, allowing diverse controller implementations to handle issues appropriately.
- Employing interfaces for core entities (e.g., accommodations, rooms, users) to facilitate easy expansion of system capabilities.

ii. Robust Error Management:

- Anticipating and preparing for unexpected scenarios and user inputs.
- Leveraging exception mechanisms to enhance system flexibility and universality.
- Avoiding direct error message output in methods to ensure compatibility with various controller types.

iii. Performance and Scalability:

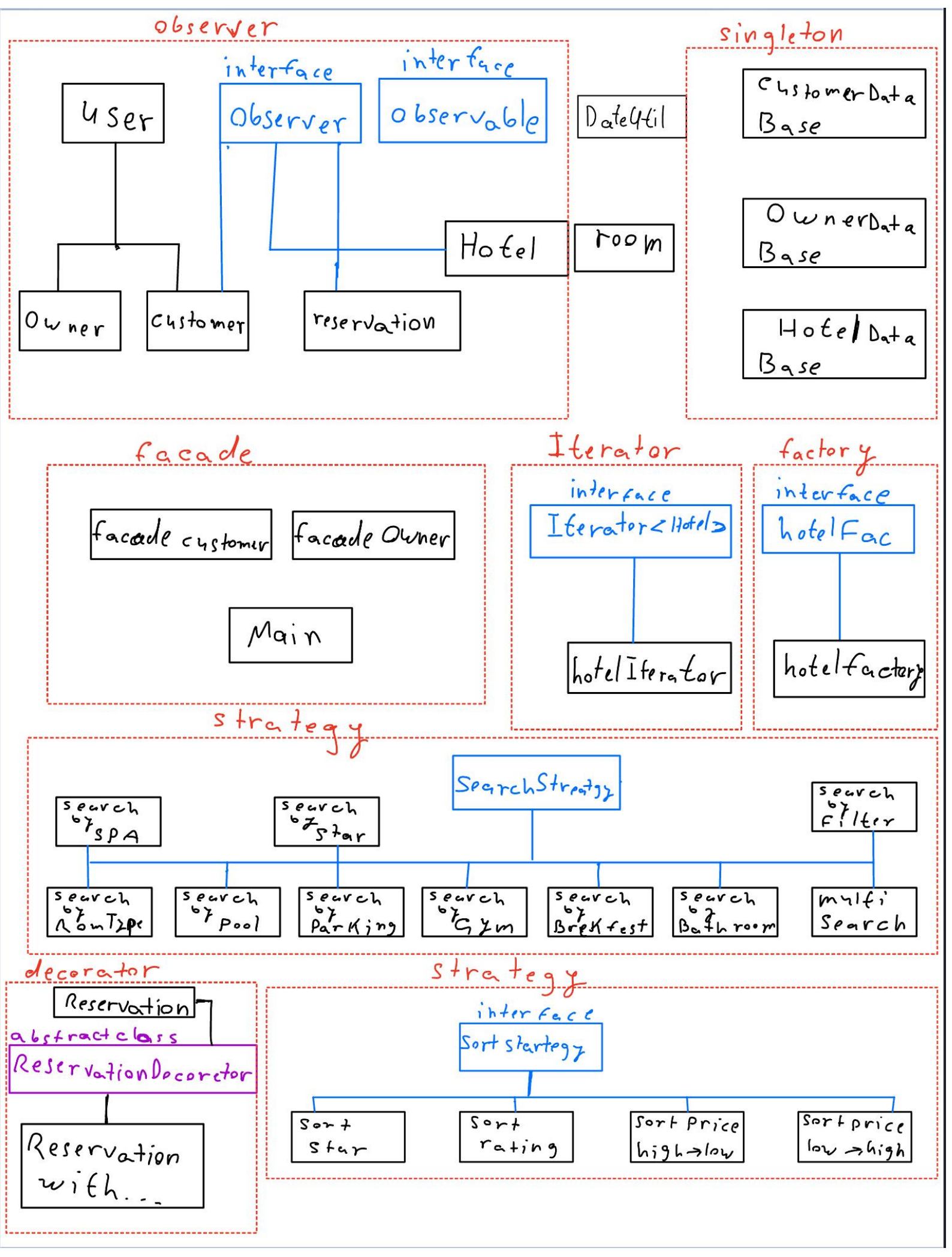
- Optimizing for large-scale operations involving numerous cities, properties, rooms, and concurrent users.
- Implementing efficient search algorithms that filter progressively (city > hotel > room) to minimize unnecessary computations.
- Enabling properties to manage room suggestions, potentially allowing for temporary reservations to prevent conflicts.

- Using interfaces to support system expandability without compromising performance.

iv. Enhanced User Interface:

- Developing a facade (façade owner class and façade customer class) that prioritizes user-friendly interactions.
- Creating intuitive menu structures and navigation flows.
- Providing clear, concise instructions for user inputs.
- Ensuring output is easily digestible and well-formatted for users.

On the next page we can see a diagram describing the classes and the relationship between them, and the design patterns


observer
interface
Observer
interface
observable
User
Owner
customer
reservation
Hotel
room
DateUtil
singleton
CustomerDataBase
OwnerDataBase
HotelDataBase
facade
facade customer
facade Owner
Main
Iterator
interface
Iterator<Hotel>
hotelIterator
factory
interface
hotelFac
hotelfactory
strategy
SearchStrategy
search by SPA
search by Star
search by Filter
search by RoomType
search by Pool
search by Parking
search by Gym
search by Breakfest
search by Bathroom
multi Search
decorator
Reservation
abstract class
ReservationDecorator
Reservation with...
strategy
interface
Sort startegy
Sort Star
Sort rating
Sort price high→low
Sort price low→high

2) Modifiers in Java:

private

i. Accessible only within the same class
ii. Highest encapsulation.

Example: private int ratingCounter; this interger from the constructor of class Hotel. with its help you can calculate the average rating the hotel received by counting the number of ratings. This data is not critical for anyone and therefore access to it is not granted outside the class

public

i. Accessible from anywhere
ii. Lowest encapsulation

Example: public static int convertDateToDays(String date). This method belong to DataUtil class. There are many methods from other classes that call this method because it performs a conversion between a date and a number, so it is public.

protected

i. Accessible within same package and subclasses
ii. Balances encapsulation and inheritance

Example: protected String password; this string token from user's constructor. The password of the user is personal. Customer ot owner extends user therefore its protected and not private.

final

i. Variables: Cannot be reassigned
ii. Methods: Cannot be overridden
iii. Classes: Cannot be inherited

Example: protected final Long id; this long token from user's constructor id is something final and does not change therefore its final

static

i. Belongs to class, not instances
ii. Accessible without object creation

Example: public static int convertDateToDays(String date). This method belong to DataUtil class.By making it static, you can call this method directly on the class without needing to create an instance of the class. This is convenient for utility methods that are frequently used.

3) Abstract Classes, Interfaces, Inheritance, and Polymorphism
(The benefits these concepts bring to object-oriented programming included)

Abstract Classes
Abstract classes are used as a base for other classes. They cannot be instantiated on their own and are designed to be extended by other classes**.** in my project ReservationDecorator is abstract class . the reservation filters such a reservationWithBreakfast is extend from the abstract class.

```
class ReservationWithLunch extends ReservationDecorator {
```

Inheritance

Inheritance is a fundamental concept in OOP that allows a class to inherit properties and methods from another class. Here are the key points about inheritance:

Basic Concept:

i. A class (subclass or derived class) can inherit attributes and methods from another class (superclass or base class)
ii. Represented by the "is-a" relationship

Types of Inheritance:

i. Single inheritance: A subclass inherits from one superclass
ii. Multilevel inheritance: A chain of inheritance (A -> B -> C)
iii. Hierarchical inheritance: Multiple classes inherit from one superclass

In my program there is a few relationships of inheritance:

i. Owner and customer extends user- they have the same fields and basic methods such user. So its promote reuse code and supports the creation of hierarchical classifications

ii. ReservationDecorator (abstract class) extends reservation- the reservation with filters is kind of reservation with unique adds so the abstract class which is actually the search pattern with filters inherits from the reservation and from it inherits the filter classes . in this way it will be much easier to make a changes in the future.

iii. `class ReservationWithBreakfast extends ReservationDecorator`

interface

interface in Java is a contract that specifies a set of abstract methods that a class must implement. It's a powerful tool for achieving abstraction and multiple inheritance of type. Here are the key points about interfaces:

Key Characteristics:

i. All methods are implicitly public and abstract (prior to Java 8)
ii. All variables are implicitly public, static, and final
iii. Cannot be instantiated directly

Multiple Inheritance:

i. Java doesn't support multiple inheritance of classes, but does for interfaces
ii. A class can implement multiple interfaces

Usage:

i. Defines a contract for classes
ii. Achieves abstraction
iii. Supports polymorphism

In my program is many interfaces:

The observer and observable that let reservation and customer contact and send messages.

SearchStrategy and SortStrategy is interfaces of all class that let make search and sort in my program. The use of interface let us make re code and save the strucrt of the search and sort.

HotelFac is interface of hotelFactory its help us to :

Flexibility: Using an interface allows for different implementations of hotel creation. While currently there's only one implementation (HotelFactory), this design allows for easy addition of other factory implementations in the future if needed.

Loose coupling: It helps in reducing dependencies between classes. Classes that use hotel creation can depend on the HotelFact interface rather than a concrete HotelFactory class, making the system more modular and easier to maintain.

Polymorphism: It enables polymorphic behavior. Different implementations of HotelFact can be used interchangeably where the interface is expected.

4)

singleton

i. Definition and Role:

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It's used when exactly one object is needed to coordinate actions across the system.

ii. Implementation:

In my project, the Singleton pattern is implemented in three classes: CustomerDataBase, HotelDatabase, and OwnerDatabase. This allows the system to search for a hotel, customer or owner without missing any information. Of course, every deletion or addition operation also affects the database.

iii. Benefits and Disadvantages:

Benefits:

1. Ensures a single instance, which is useful for managing global resources.
2. Provides a global access point to that instance.
3. The instance is only created when it's first requested (lazy initialization).

Disadvantages:

1. Can make unit testing more difficult.
2. Violates the Single Responsibility Principle as the class manages its own creation and lifecycle.

3. Can be overused, leading to unnecessary global state in the application.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

The singleton provides a consistent way to access global resources across the application. But its also can lead to tight coupling if overused, making it harder to reuse components independently.

Maintainability:

The singleton centralizes management of shared resources, making it easier to update or modify their behavior.But it can create hidden dependencies in the code, making it harder to understand and maintain.

Façade

i. Facade Definition and Role:

The Facade pattern provides a simplified interface to a complex subsystem. It acts as a high-level interface that makes the subsystem easier to use.

ii. Implementation:

In my project, the Facade pattern is implemented through the FacadeOwner and FacadeCustomer classes. These classes provide simplified interfaces for owners and customers to interact with the hotel booking system.

iii. Benefits and Disadvantages:

Benefits:

1. Simplifies the interface for clients, making the system easier to use.
2. Reduces dependencies between clients and subsystem components.
3. Promotes loose coupling between the client and the subsystem.
4. Provides a layer of abstraction, hiding complex subsystem details.

Disadvantages:

1. Can become a "god object" coupled to all classes of the subsystem.
2. May introduce an additional layer of indirection, potentially impacting performance.
3. Can violate the Single Responsibility Principle if not careful.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

In one hand the facade can be reused across different parts of the application that need to interact with the same subsystem. But on the other hand if not designed carefully, it might limit the ability to reuse individual components of the subsystem independently.

the FacadeOwner and FacadeCustomer classes provide clear, role-specific interfaces to the hotel booking system, simplifying interactions for both hotel owners and customers. This separation of concerns improves the overall structure and maintainability of the system.

Decorator

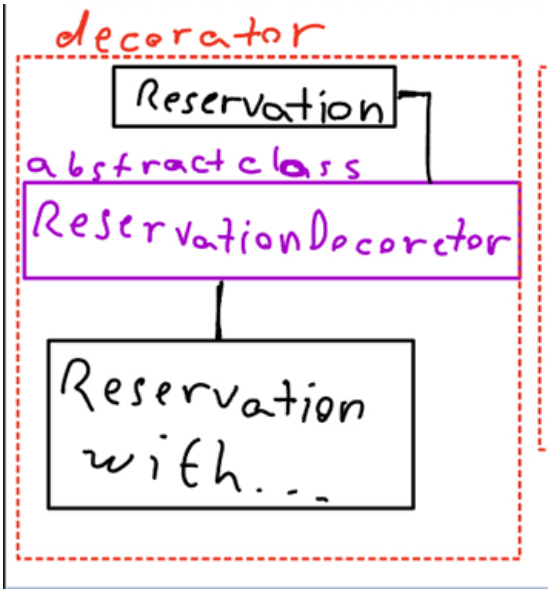
i. Decorator Definition and Role:

The Decorator pattern allows behavior to be added to individual objects dynamically without affecting the behavior of other objects from the same class. It's used to extend or alter the functionality of objects at runtime.In my project, the Decorator pattern is used to add additional features or costs to a basic reservation, such as breakfast, lunch, dinner, or additional guests, without modifying the core Reservation class.

ii. Implementation:

The implementation consists of:

1. A base Reservation class
2. An abstract ReservationDecorator class
3. Concrete decorator classes like ReservationWithBreakfast,ReservationWithLunch, etc.



iii. Benefits and Disadvantages:

Benefits:

1. Flexibility: New features can be added without modifying existing code.
2. Open/Closed Principle: The pattern adheres to the open/closed principle, allowing for extension without modification.

   Disadvantages:

1. Complexity: Can lead to many small classes, which might be harder to understand.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

- The Decorator pattern promotes code reuse by allowing functionality to be shared across multiple decorator classes.
- New features can be added by creating new decorator classes without modifying existing code.

However, overuse of the pattern can lead to a proliferation of small, similar classes, which might make the codebase harder to navigate and understand for new developers.This pattern particularly helps improve maintainability in systems where features need to be combined in various ways, as it allows for flexible combination of behaviors without creating a class explosion through subclassing.

Iterator

i. Iterator Definition and Role:

The Iterator pattern provides a way to access elements of an aggregate object sequentially without exposing its underlying representation. It solves the problem of traversing complex data structures (like nested collections) in a uniform way, regardless of their specific implementation.In my project, the Iterator pattern is used to traverse a complex structure of hotels organized by country and city, allowing clients to iterate over all hotels without needing to understand the nested structure.

ii. Implementation:

The implementation consists of:

1. An `Iterator<Hotel>` interface (from Java's standard library)
2. A concrete `HotelIterator` class implementing this interface.

Key code snippet:

```
class HotelIterator implements Iterator<Hotel> {
  private Iterator<Map<String, List<Hotel>>> countryIterator;
  private Iterator<List<Hotel>> cityIterator;
  private Iterator<Hotel> hotelIterator;

  public HotelIterator(Collection<Map<String, List<Hotel>>> hotels) {
    countryIterator = hotels.iterator();
    cityIterator = Collections.emptyIterator();
    hotelIterator = Collections.emptyIterator();
  }

  @Override
  public boolean hasNext() {
    // Logic to check if there are more hotels
  }

  @Override
  public Hotel next() {
    // Logic to get the next hotel
  }
}
```

iii. Benefits and Disadvantages:

Benefits:

1. Simplifies client code: Clients can traverse complex structures without understanding their internals.
2. Supports multiple traversals: Different iterators can traverse the same collection concurrently.

Disadvantages:

1. Increased complexity: For simple collections, using an iterator might be overkill.
2. Performance overhead: There's a slight performance cost due to the additional abstraction layer.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

- The Iterator pattern promotes code reuse by separating the traversal logic from the collection implementation.
- The same iterator can be used for different types of collections with similar structures.

In this specific implementation, the iterator helps manage the complexity of traversing a nested structure (countries -> cities -> hotels) transparently to the

client. This separation of concerns makes the code more maintainable, as changes to the data structure can be localized to the iterator implementation.

Strategy

i. Strategy Definition and Role:

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

ii. Implementation:

In this project, the Strategy pattern is used to implement different sorting algorithms for a list of hotels. This allows the sorting behavior to be selected at runtime without modifying the client code that uses the sorting.

The implementation consists of:

1. A SortStrategy interface
2. Concrete strategy classes implementing different sorting algorithms

Key code snippets:

```
public interface SortStrategy {
    void sort(List<Hotel> hotels);
}

public class SortByStar implements SortStrategy {
    @Override
    public void sort(List<Hotel> hotels) {
        hotels.sort(Comparator.comparingInt(Hotel::getStars).reversed());
```

iii. Benefits and Disadvantages:

Benefits:

1. Flexibility: Allows for easy swapping of algorithms at runtime.
2. Encapsulation: Each sorting algorithm is encapsulated in its own class.
3. Open/Closed Principle: New sorting strategies can be added without modifying existing code.

Disadvantages:

1. Increased number of classes: Can lead to a proliferation of strategy classes in the system.
2. Communication overhead: If the strategies need to share data with the context, it might lead to communication overhead.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

- The Strategy pattern promotes code reuse by allowing sorting algorithms to be shared across different parts of the application.
- New sorting strategies can be easily added and reused without modifying existing code.

Maintainability:

- Improves maintainability by isolating sorting algorithms in separate classes.
- Changes to one sorting strategy don't affect others or the client code using them.
- Makes it easier to test individual sorting strategies in isolation.

In my code, the Strategy pattern allows for easy addition of new sorting methods for hotels. If a new sorting criterion is needed (e.g., sort by amenities), a new strategy class can be added without modifying existing code.

Factory

i. Factory Definition and Role:

The Factory pattern is a creational pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It's used to decouple object creation from the code that uses the object.

In my project, the Factory pattern is used to create Hotel objects, encapsulating the complex creation process and allowing for potential extension of hotel types in the future.

iii. Implementation:

In my project, the Factory pattern is used to create Hotel objects, encapsulating the complex creation process and allowing for potential extension of hotel types in the future.

The implementation consists of:

1. A HotelFact interface defining the creation method
2. A concrete HotelFactory class implementing this interface

iii. Benefits and Disadvantages:

Benefits:

1. Encapsulation: The complex object creation process is encapsulated in a single place.
2. Flexibility: It's easy to introduce new types of hotels without changing existing code.
3. Separation of concerns: The client code is decoupled from the specifics of object creation.

Disadvantages:

1. Increased complexity: For simple object creation, using a factory might be overkill.
2. Difficulty in subclassing: If the factory method is in a base class, all subclasses must override it.

iv. Impact on Code Reuse and Maintenance:

Code Reuse:

- The Factory pattern promotes code reuse by centralizing the object creation logic.
- It allows for easy reuse of creation code across different parts of the application.

Maintainability:

- Improves maintainability by isolating the creation logic, making it easier to update or extend.
- Changes to the hotel creation process can be made in one place without affecting client code.
- Makes it easier to add new types of hotels or modify existing ones.

In my program, the Factory pattern allows for a structured way to create hotels with various attributes and rooms. If new types of hotels or creation processes are needed in the future, they can be easily added by extending the HotelFactory or creating new factory classes implementing the HotelFact interface.

5) Generics and Collections:
   a. Generics are useful because they allow us to write code once and reuse it with differentdata types. The HashMap and ArrayList data structures in Java are implemented with generics. This structures allowed me to have information such as a list of hotels, rooms, customers, owners and reservations without realizing the data structures from scratch.
   b.

```
 private Map<Long, Customer> customers;

private Map<String, Map<String, List<Hotel>>> hotels;
```

hash map of hash map of hotels to make the search more efficient

```
    private List<Reservation> reservations = new ArrayList<>();
    private Map<Long, OwnerHotel> owners;
```

6) Exception Handling Strategies
   a. In my program there is a lot of inputs get from the customer or the owner. In addition the search and the build of the data base based on the inputs. So its important to ensure that the inputs are exactly what the development mean. My strategy of exption handling is not to let the program fail because in correct input Rather, printing an incorrect input and re-calling the method in order to correct the input. In this way, I enable correct data entry and do not crash the program on improper input, even if proactively for example:

```
private Customer getIn() {
        System.out.println("Enter your id:");
        Long id = scanner.nextLong();
        scanner.nextLine();

        Customer customer =
CustomerDataBase.getInstance().getCustomer(id);
        if (customer == null) {//if the customer is not registered
            System.out.println("You are not registered, please enter
your name and password to register");
            System.out.println("Enter your name:");
            String name = scanner.nextLine();
            System.out.println("Enter your password:");
            String password = scanner.nextLine();
            customer = new Customer(name, id, password);
            CustomerDataBase.getInstance().addCustomer(customer);
            customer.login(id, password);
            if (customer.isConnected()) {
                System.out.println("You are connected.");
```

```
            } else {
                System.out.println("Your access failed. Please try
again.");
                getIn();//if the access failed, the customer should try
again
            }
```

b)

this method add rooms to the hotel.the customer search the hotel and make the reservation according the room type :standard,e.g, premium and etc. so In order to enable a proper search, you must make sure that in the creation of the hotel there is a room that fits properly

```
 public void addRooms(String roomType, int quantity, double price, int
bedCount, boolean hasBalconey, boolean hasPrivateBathroom) {
        if  (!roomType.equals("Standard") && !roomType.equals("e.g") &&
!roomType.equals("Premium") && !roomType.equals("etc")) {
        System.out.println("Invalid room type");
            return;
```

7) Code Optimization and Effciency:

a. Strategy to optimize the code for better performance and efficiency

i. Having all the hotels in Hash Map's Hash Map makes it possible to optimize the hotel search process. This configuration is based on the assumption that it is reasonable to define the country and the city as mandatory fields for the search and in return we dramatically optimize the search and do not go through a list of all the hotels. The difference will be mainly when the system will hold a very large amount of hotels

ii. Finding an available room is done as follows - the rooms are kept by the hotel in lists according to room types. Each room has its own list of occupied dates, and this is how we actually find an available room and call the reservation table with that room. Although in the process we have a loop within a loop, but this is done after that we narrowed the search range by room type. In addition, we saved memory since a hotel room does not need to have a data structure among 365 cells that will hold the information whether the room is available or not a year ahead

b. I think that for a hotel reservation system importance should be given to the clarity of the code and the maintenance methods. Because the calculations that the system is required to perform are not very complex and the computer can perform them without being less efficient but in such a way that the user will not feel the differences in the running time. I do think it is important to analyze key bottlenecks in the code and prioritize optimization over readability and maintenance such as managing searches on the customer, owner and hotel singletons

8) Testing and debugging

a. My approach to code testing was testing while coding. At the same time as the departments, I managed a "breathing main" where I performed dedicated tests for each and every method, of course for each method I examined edge cases

b. The size and complexity of the project was a major challenge in checking the text. Lots of methods call each other and when a code crashes or you get a wrong answer it is very difficult to locate the source of the problem, using a debugger sometimes helped to locate the source of the problem. Another challenge is the amount of input I receive from the user during the run, which requires me to perform a lot of tests and safety nets in order to prevent the program from crashing when entering incorrect input.

c.

Patterns often lead to more modular, decoupled code that's easier to unit test. Patterns like MVC separate components, enabling isolated testing. Patterns provide consistent structures, allowing for standardized test approaches. Better . Modular code from patterns often results in improved test coverage. Patterns like Façade can simplify testing of subsystem . Some patterns (e.g., Singleton) may require specific performance tests.