**Name:** Daniel Samson   
**ID:** 211358486  
**Email:** [danielsamson10@gmail.com](mailto:danielsamson10@gmail.com)  
 **General Explanation**

The hotel booking system facilitates user interactions through a structured MVC (Model-View-Controller) architecture, ensuring clear separation of concerns and promoting modularity. The system primarily consists of three main components: **Controller**, **System Booking (Model)**, and **View**.

**Controller:** Acts as the central hub that manages user requests and delegates tasks to other components based on user input. It ensures that the system responds appropriately to the user actions and handling inputs.

**System Booking (Model):** Represents the backend logic responsible for managing users and managers, hotels, bookings, and notifications and more. It encapsulates the core functionalities and data management operations, ensuring data integrity and consistency throughout the system's lifecycle and by that its facade to all the classes.

**So how it works?**  
We start with menu to ask the user for log in/new user. After that, he gets a new menu based on what he chooses. For example, by log in user he gets user options for doing any action, and by log in manager he gets manager options for doing any action, and then by choosing based on the menu, he gets the action he wants to do.  
  
**System Booking** interacts with all the classes here are several key classes:

* **Person:** Represents an individual user/manager of the system, encapsulating user-related data such as username, password, and contact information.
* **User:** Users can perform actions like making/paying/cancel reservation, receiving notifications, searching for hotel/specific room, and viewing wish list and former orders.
* **Manager**: Represents management of hotels, allowing them to manage hotel, room information and creation, send message to users for event etc.
* **Hotel:** Represents a hotel entity, such as hotel name, location, rooms and amenities.
* **Room:** Represents a room entity, such as room name, availability and amenities.  
  The Room class manages availability and provides methods to check availability and make reservations.

**View (UI):** Provides the user interface layer, presenting information to users. The **View** is designed to be intuitive and user-friendly, guiding users through the booking process and displaying relevant information such as menu, available hotels, booking details, and notifications.

The system operates on principles of encapsulation, inheritance, and polymorphism to enhance code reusability, maintainability, and scalability.  
Each component interacts through well-defined interfaces, adhering to the MVC pattern to facilitate easy modification and extension.

**How Classes Interact:**

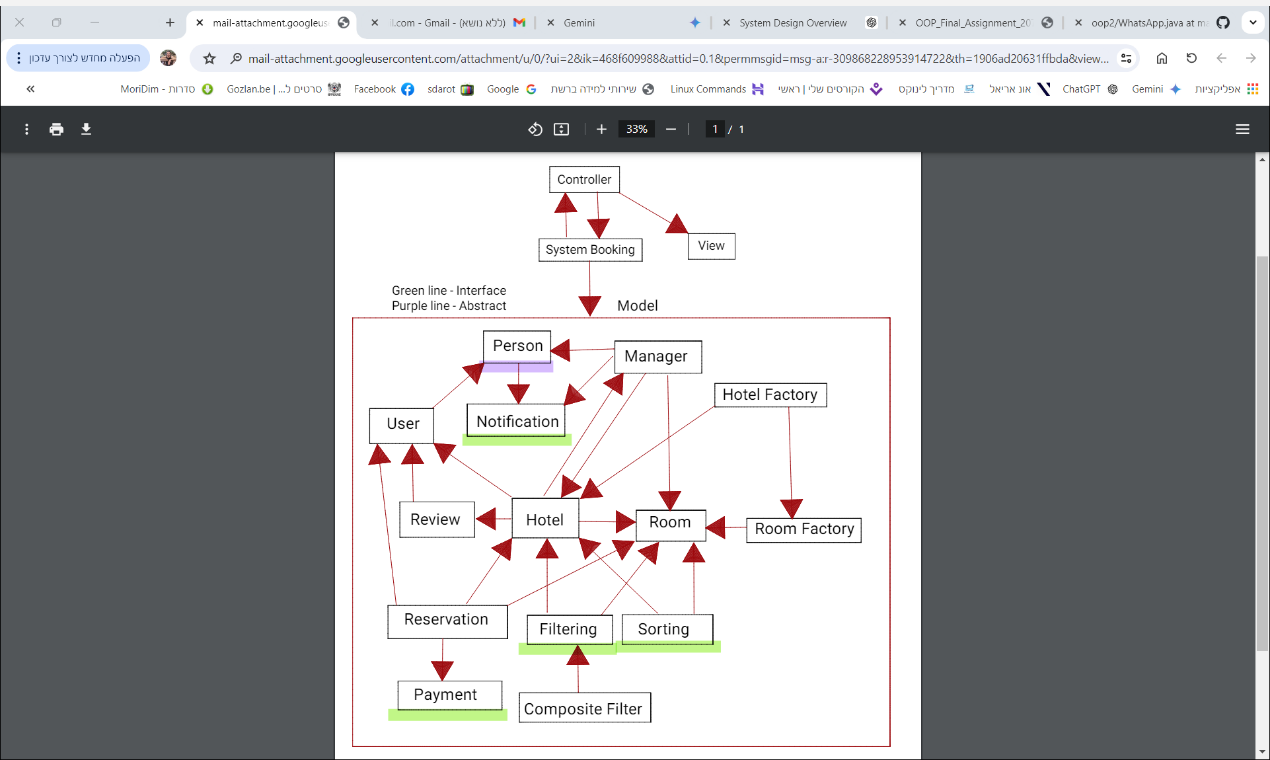
**How it works from above:**

* **User (user/manager) Interaction:**When a user interacts with the system, their input (e.g., search for hotels, make a reservation) is captured by the Controller. The Controller sends this input to the System Booking.
* **Controller Processing:**The Controller processes the input and invokes appropriate methods on the System Booking. For instance, if a user searches for available hotels, the Controller calls the search Hotels method on System Booking.
* **System Booking Operations:** System Booking performs the requested operations, interacting with classes like Hotel and Room and Reservation. It retrieves data, processes the data, and ensures that actions follow rules and constraints.
* **View Update:**  
   Once the operation is complete, System Booking returns the results to the Controller. Then, Controller updates the View with the relevant information, such as displaying available hotels or confirming a reservation.

**How it works from below:  
 - In the system booking, I interact with all the other classes. Here are some of them:**

* **Filtering:** The Filtering class handles the logic for applying various filters for reaching to the list of hotels. Filters can be based on criteria such as price range, amenities, star rating, location, etc.
* **Sorting:** The Sorting class handles the logic for sorting the list of hotels based on different criteria such as price, rating, distance, etc.
* **Reservation**: The Reservation class encapsulates the details of a booking, including user information, hotel details, room type, booking dates, and payment status. It ensures that reservations adhere to availability constraints and business rules.
* **Review:** The Review class manages user reviews for hotels, encapsulating review details such as user information, rating, comments, and date.
* **Notification:** The Notification class manages notifications sent to users and managers regarding booking statuses, payment confirmations, and other important updates.
* **Payment**: The Payment class handles payment processing, encapsulating payment details such as payment method, amount, and transaction status.

**iii. What is the relationship between the classes:**The MVC pattern separates the system (code) into three main components, for a clear separation of concerns:  
**Model (System Booking):** Represent the system's data and logic. User, Manager, Hotel, Booking, Room, Review, Notification, and Payment are part of the Model layer. Also using façade and singleton pattern.  
**View:** Manages the presentation layer, displaying data to users.  
**Controller**: Handles user input, interacts with the Model to retrieve and manipulate data, and updates the View accordingly. **Creating Room and Hotel:** using factory pattern.  
**Filtering and Sorting:** both using the strategy pattern.  
**Composite Filtering:** using composite pattern.  
**Date:** date inside Room using iterator pattern.  
**Notification:** using observer pattern.

****

**2. Modifiers:**

* **Private:** restricts access to the members (variables, methods) so that they can only be accessed within the same class.
* **Protected:** allows access to members within the same class and subclasses.
* **Public:** allows members to be accessible from any other class.
* **Final:** indicates that the member or class cannot be modified once it is initialized. For methods, it means they cannot be overridden; for classes, it means they cannot be subclassed.
* **Static:** indicates that the member belongs to the class itself rather than to any particular instance of the class.

**Example:**

* **Private:** In User class, user credentials should be access only from inside the class, and can be controlled by getting access by getters and setters.  
  public class User extends Person{  
  private List<Reservation> pastOrders;  
  public List<Reservation> getPastOrders() {

return this.pastOrders;

}  
public void addPastOrder(Reservation reservation) {

if (reservation.getStatus().toString().equals("CONFIRMED")) {

String message = String.format("Reservation for room %d from %s to %s has been CONFIRMED.",

reservation.getRoom().getRoomId(), reservation.getCheckInDate(), reservation.getCheckOutDate());

sendNotification(message);

pastOrders.add(reservation);

} else {

System.out.println("Order not completed yet");

}

}}  
 public void removePastOrder(Reservation reservation) {

if (reservation.getStatus().toString().equals("CANCELLED")) {

pastOrders.remove(reservation);

String message = String.format("Reservation for room %d from %s to %s has been CANCELLED.",

reservation.getRoom().getRoomId(), reservation.getCheckInDate(), reservation.getCheckOutDate());

sendNotification(message);

} else {

System.out.println("Order not CANCELLED yet");

}

}}

* **Protected:** In Person class, user credentials such as name and password are marked as protected to ensure they can be accessed only from inside of the class and subclasses.  
  public abstract class Person {

protected String name;

protected String password;

// Getter and Setter methods

public String getName() {

return name;

}  
public void setName(String name) {

this.name = name;

}

public void setPassword(String oldPassword, String newPassword) {

if (oldPassword.equals(this.password)) {

this.password = newPassword;

}

}  
public String getPassword() {

return password;

}}

* **Public:** every function that we want to give access to from anywhere, like in person class, we want to give access to everyone that can get the name of that person.  
  public abstract class Person{  
  public String getName() {

return name;

}  
public void setName(String name) {

this.name = name;

}}

* **Final**: what we don’t want to change, in class person once we give id to person we don’t want it to change.public abstract class Person{  
  protected final long id;  
  }
* **Static:** when I want a function or member to belong to the class and not to an instance, in the person class I want to give to each person a unique id then I make the count static so it will belong to the class and each time an instance created I add one to it give him a unique number.public abstract class Person{  
  public static long count = 0;  
  public Person{  
  this.id = count++;  
  }

**3. Abstract Classes, Interfaces, Inheritance, and Polymorphism:**

* **Abstract:**I used an abstract Person Class:

This class serves as a blueprint, encapsulating attributes and behaviors common to both User and Manager entities within my system. This could include essential details like name, email address, an ID number and more.

It enforces a common structure, ensuring that both User and Manager have essential attributes like name and email.

It might declare an abstract method (methods without implementation) that must be defined in the subclasses. These methods represent operations specific to a person, but with varying implementations for different user types.

* **Inheritance:**   
  Inheritance allows a class to inherit properties and behaviors from another class, promoting code reuse and hierarchical relationships. For example, the User and Manager classes extends the Person class to inherit common person properties and behaviors like:

public abstract class Person {  
protected String name;  
protected String email;  
}  
public class Manager extends Person{}  
public class User extends Person{}  
  
The User and Manager classes inherits name and email properties from Person.   
Inheritance facilitates code reuse and supports hierarchical relationships among classes.

* **Polymorphism:**   
  Polymorphic Behavior: Because User and Manager both inherit from Person, I can create an array or list of Person objects. This list can then hold instances of both User and Manager.

Dynamic Method Dispatch: When I call a method (like the hypothetical greet()) on a Person object in the list, Java uses dynamic method dispatch at runtime to determine the correct implementation to use based on the actual object's type (User or Manager). This allows for flexible behavior based on the object at hand.

Benefits in my code:

Code Reusability: By defining common attributes and behaviors in Person, I avoid duplicating that code in User and Manager.

Maintainability: Changes made in Person (e.g., adding a new common attribute) automatically propagate to both User and Manager, simplifying maintenance.

Extensibility: The structure allows to easily add new subclasses (e.g., Admin) that inherit from Person and provide specialized functionality.

Polymorphic Flexibility: I can treat User and Manager objects uniformly as Person objects in certain contexts, leading to more flexible code that can work with different types of people.

4. **Design pattern:**

* **Singelton:**

**Definition and Role:** The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance. It is useful when exactly one object is needed to coordinate actions across the system.  
**Implementation:**  
public class SystemBooking {  
private static SystemBooking instance;  
private SystemBooking(){// Private constructor  
public static SystemBooking getInstance(){  
if (instance == null){  
instance = new SystemBooking();  
}  
return instance;  
}

The SystemBooking class ensures there is only one instance of the booking system throughout the application. The getInstance() method provides global access to this instance, ensuring centralized management of system-wide operations.

**Benefits:** Centralized instance management simplifies access to shared resources and ensures consistent behavior across the application. It promotes resource efficiency by preventing multiple instances and facilitates lazy initialization.

**Disadvantages:** Singleton pattern can introduce tight coupling between classes, making unit testing challenging. It also limits scalability in distributed environments where multiple instances might be necessary.  
**Impact on Code Reuse and Maintenance:** The Singleton pattern can greatly enhance code reuse by providing a single, reusable instance of a class and centralizing control over its instantiation and access. However, care must be taken to manage its impact on maintainability.

 **Factory:**

**Definition:** The Factory pattern defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.

**Role:**

Implementation:   
public List<Hotel> filterHotels(List<Hotel> hotels, List<String> filter, List<String> amenitiesHotel, List<String> amenitiesRoom, Date start, Date finish, double priceMin,

double priceMax, int rating, String location) {

boolean b = false;

this.filter = new CompositeFilter();

for (String f : filter) {

switch (f.toLowerCase()) {

case "amenities to hotel":

if (amenitiesHotel != null) {

Filtering filtering = new FilterByAmenitiesHotel(amenitiesHotel);

this.filter.addFilter(filtering);

b = true;

break;

}

case "amenities to room":

if (amenitiesRoom != null) {

Filtering filtering = new FilterByAmenitiesRoom(amenitiesRoom);

this.filter.addFilter(filtering);

b = true;

break;

}

case "date":

if (start != null && finish != null) {

Filtering filtering = new FilterByDate(start, finish);

this.filter.addFilter(filtering);

b = true;

break;

}

case "price":

if (priceMin > 0 && priceMax >= priceMin) {

Filtering filtering = new FilterByPrice(priceMin, priceMax);

this.filter.addFilter(filtering);

b = true;

break;

}

case "rating":

if (rating > 0) {

Filtering filtering = new FilterByRating(rating);

this.filter.addFilter(filtering);

b = true;

break;

}

case "location":

if (location != null) {

Filtering filtering = new FilterByLocation(location);

this.filter.addFilter(filtering);

b = true;

break;

}

default:

break;

}

}

if (b) {

return this.filter.filter(hotels);

}

return null;

}

**Benefits:** Encapsulates object creation logic, promoting flexibility.

Enhances code readability and maintainability.

Makes the system more scalable by allowing easy addition of new types of objects.

**Disadvantages:** Can introduce complexity if overused.

Factory methods can become cumbersome if there are many types to instantiate.

**Code Reuse:** The Factory pattern centralizes the creation logic, making it easier to reuse and extend.

**Maintenance:** Simplifies maintenance by encapsulating the instantiation process, making the system more adaptable to change. The use of factories can improve maintainability by decoupling the client code from the specific classes being instantiated.

* **Strategy:**

**Definition:** The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from clients that use it.

**Role:** In the hotel booking system, the Strategy pattern is used for sorting hotel data based on different criteria such as price, rating, or location. This allows the sorting logic to be changed dynamically at runtime without altering the clients that use the sorting algorithms.

**Implementation:**

public List<Hotel> sortHotels(String sort) {

switch (sort.toLowerCase()) {

case "id":

this.sort = new SortById();

break;

case "price":

this.sort = new SortByPrice();

break;

case "rating":

this.sort = new SortByRating();

break;

case "rating and price":

this.sort = new SortByRatingAndPrice();

break;

case "room size":

this.sort = new SortByRoomsInHotel();

break;

case "top review":

this.sort = new SortByTopReview();

break;

default:

break;

}

this.sort.sort(hotels);

return hotels;

}

**Benefits:**

Flexibility: Easily switch between different sorting algorithms at runtime.

Encapsulation: Encapsulates sorting algorithms, promoting separation of concerns.

Maintainability: Simplifies the addition of new sorting criteria without modifying existing code.

**Disadvantages:**

Overhead: May introduce slight overhead due to additional object creation and method calls.

Complexity: Can add complexity if there are many sorting strategies to manage.

**Impact on Code Reuse and Maintenance:**

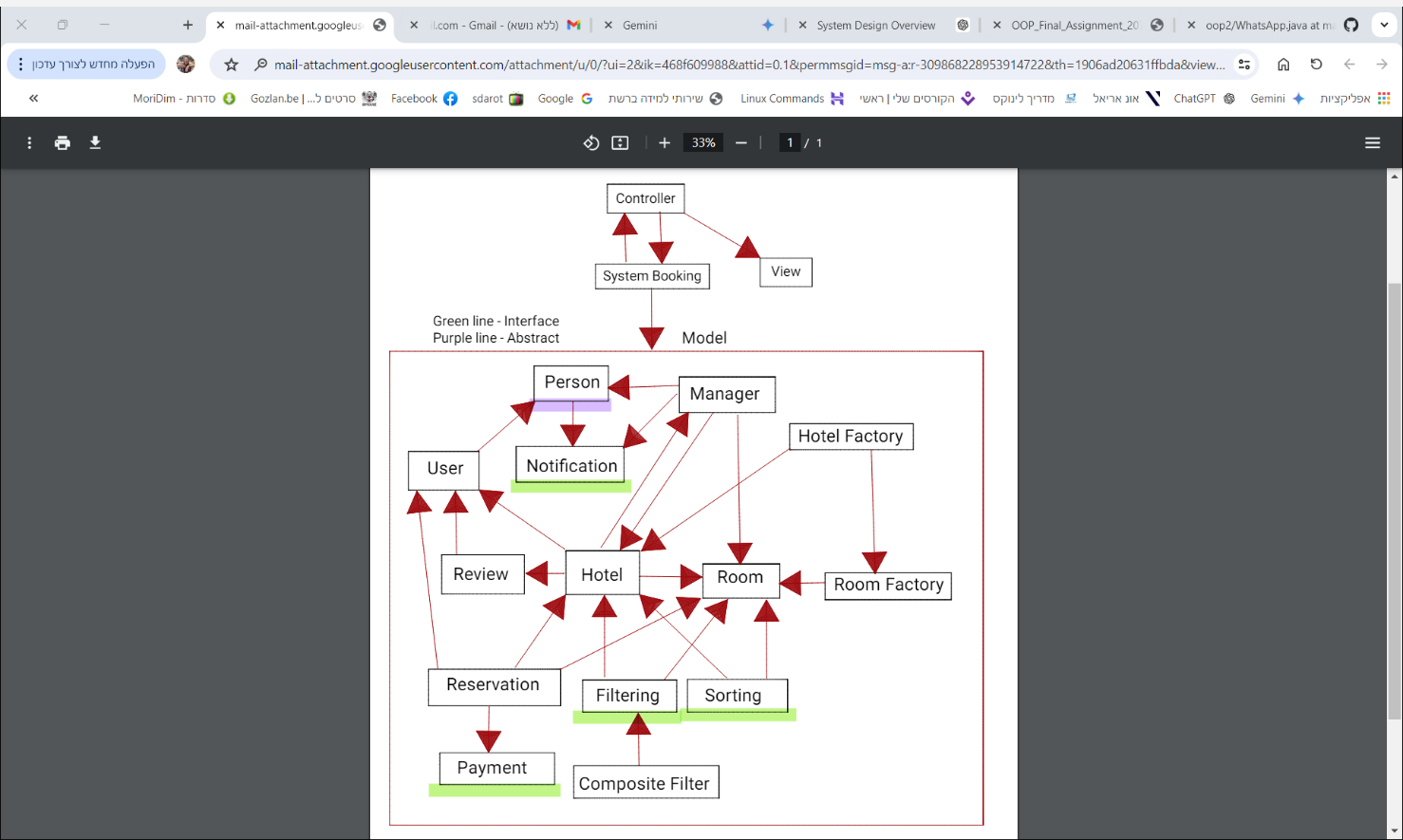
**Code Reuse:** The Strategy pattern promotes code reuse by allowing the same sorting logic to be applied in different contexts without duplicating code. New sorting strategies can be implemented and integrated easily without changing existing client code.

**Maintainability:** Enhances maintainability by encapsulating sorting algorithms in their own classes. This makes it easy to add, modify, or remove sorting strategies as needed. The use of well-defined interfaces ensures that changes to one sorting strategy do not impact others, thus promoting a modular and maintainable codebase.

* **Facade:**

**Definition:** The Facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

**Role:** In the hotel booking system, the Facade pattern is implemented using the SystemBooking class. The SystemBooking class acts as a single-entry point for clients to interact with various subsystems such as user management, hotel management, booking operations, filtering, and notifications. This simplifies client interactions by exposing a unified interface and hiding the complexities of the underlying subsystems.

**Implementation:**  **here UML that show that system booking interact with all the classes and controller and view don’t interact with them.**

**Benefits:**

Simplified Interface: Provides a simplified interface to complex subsystems, making the system easier to use for clients.

Decoupling: Decouples clients from the subsystem details, allowing changes to be made to the subsystems without affecting the clients.

Code Organization: Organizes code by encapsulating interactions with multiple subsystems within a single class.

**Disadvantages:**

Single Point of Failure: The facade can become a single point of failure if it encapsulates too much logic.

Limited Flexibility: While the facade simplifies interactions, it may limit flexibility by not exposing all functionalities of the subsystems.

**Impact on Code Reuse and Maintenance:**

**Code Reuse:** The Facade pattern promotes code reuse by providing a centralized interface for interacting with subsystems. This avoids the need for clients to duplicate code for subsystem interactions.

**Maintainability:** Enhances maintainability by encapsulating subsystem interactions within a single class. This makes it easier to update or modify the underlying subsystems without impacting the client code. Additionally, by reducing the complexity of client interactions, the facade makes the system easier to understand and maintain.

* **Composite:**

**Definition and Role:  
Purpose:** The Composite pattern composes objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.  
**Problem it Solves:** It solves the problem of working with hierarchical structures of objects where individual objects and compositions of objects need to be treated uniformly.

**Implementation:**  
public interface Filtering {  
List<Hotel> filter(List<Hotel> hotels);  
}  
import java.util.ArrayList;  
import java.util.List;  
public class CompositeFilter implements Filtering {  
private List<Filtering> filters = new ArrayList<>();  
public void addFilter(Filtering filter) {  
filters.add(filter);  
}  
@Override  
public List<Hotel> filter(List<Hotel> hotels) {  
List<Hotel> filteredHotels = hotels;  
for (Filtering filter : filters) {  
filteredHotels = filter.filter(filteredHotels);  
}  
return filteredHotels;  
}}

**Benefits and Disadvantages:**

**Benefits:**

Allows clients to treat individual filters and composite filters uniformly when applying filtering operations.

Supports nested structures of filters, enabling complex filtering requirements to be managed efficiently.

Promotes code flexibility and scalability in handling varied filtering criteria.

**Disadvantages:**

Increased complexity in managing the composite structure and ensuring correct behavior across all nested filters.

Potential performance overhead when processing deeply nested structures of filters.

**Impact on Code Reuse and Maintenance:**

**Code Reuse:** The Composite pattern promotes reuse by allowing the same interface (Filtering) to be applied uniformly to individual filters and composite filters.

**Maintainability:** By encapsulating complex filtering logic into composites and individual filters, the Composite pattern simplifies maintenance and modifications to filtering behavior. It also enables the addition or removal of filters without affecting other parts of the system, enhancing maintainability.

* **Observer:**

**Definition:** The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Role:** In the hotel booking system, the Observer pattern is used to notify users and managers about booking updates, payment confirmations, and other relevant events. It ensures that multiple entities (observers) can monitor and react to changes in a subject (notification) without tightly coupling them.  
  
**Implementation:**public interface Notification {

void sendNotification(String message, Person person);

}

import java.util.Objects;

public class WhatsApp implements Notification {

@Override

public void sendNotification(String message, Person person) {

System.out.println("Sending whatsapp to " + person.getName() + " with " + person.getPhone() + " : " + message);

}

@Override

public String toString() {

return "WhatsApp";

}

@Override

public boolean equals(Object o) {

if (o == null ) return false;

if (this == o||getClass()==o.getClass()) return true;

return Objects.equals(toString(), o.getClass().toString());

}

@Override

public int hashCode() {

return Objects.hash(toString());

}

}  
  
Benefits:

Loose Coupling: Allows observers (users, managers) to react to changes in the notification subject (e.g., booking status) without being tightly coupled to it.

Flexibility: Supports dynamic relationships where subjects and observers can be added or removed at runtime.

Event-driven: Facilitates event-driven architectures where changes in one part of the system trigger actions in other parts.

**Disadvantages:**

Unexpected Updates: Observers may receive updates they are not interested in, leading to unnecessary processing or overhead.

Complexity: Managing multiple observers and ensuring proper notification can add complexity to the system.

**Impact on Code Reuse and Maintenance:**

**Code Reuse:** The Observer pattern promotes code reuse by allowing multiple observers to react to notifications from a single subject. It encapsulates the notification mechanism, making it easier to add new observers without modifying the subject.

**Maintainability:** Enhances maintainability by decoupling the notification sender (subject) from its receivers (observers). Changes in notification logic or the addition/removal of observers do not affect each other, leading to a more modular and maintainable system.

* **MVC:**

**Definition:** The MVC (Model-View-Controller) pattern separates an application into three interconnected components: the Model, View, and Controller. Each component handles specific responsibilities:

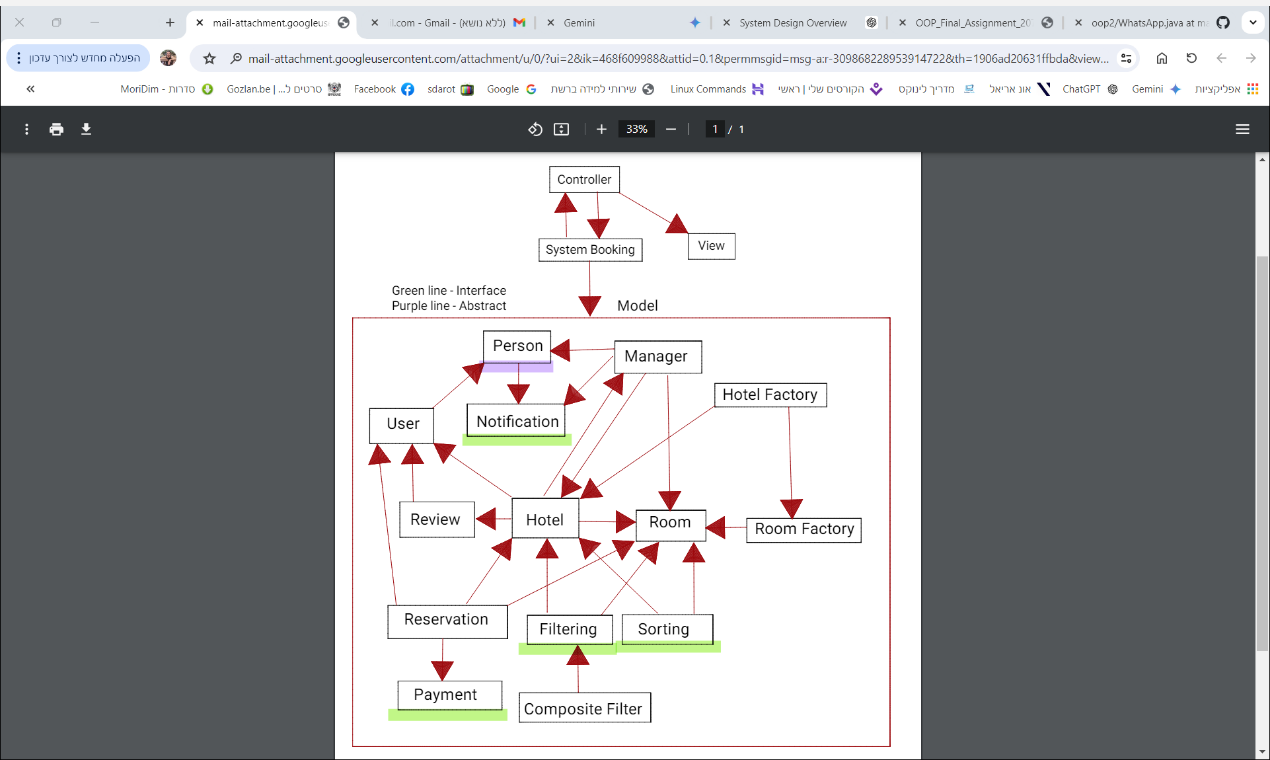
Model: Represents the application's data and business logic. It encapsulates data manipulation, validation, and state management.

View: Presents the data to the user and handles user interface interactions. It displays information from the Model and sends user commands to the Controller.

Controller: Acts as an intermediary between the Model and View. It interprets user actions (inputs) and translates them into operations on the Model. The Controller updates the View when the Model changes.

**Role:** MVC promotes separation of concerns, enhancing maintainability and scalability of applications. It isolates the presentation layer (View) from business logic (Model), allowing each component to evolve independently.

**Implementation: UML OF MVC**

****

**Benefits:**

Modularity: Separates concerns between data (Model), presentation (View), and user interaction (Controller), making components easier to maintain and extend.

Reusability: Components can be reused in different parts of the application or even in other applications, promoting code reuse.

Scalability: Supports large applications by structuring code into manageable components that can be developed and tested independently.

**Disadvantages:**

Complexity: Introducing the MVC pattern can add initial complexity, especially for smaller applications where the separation might seem overkill.

Synchronization: Ensuring proper synchronization between Model updates and View updates can be challenging, leading to potential inconsistencies if not managed carefully.

**Impact on Code Reuse and Maintenance:**

**Code Reuse:** MVC facilitates code reuse by separating concerns. For example, the same Model can be used with different Views or Controllers, depending on the application's needs.

**Maintainability:** Enhances maintainability by isolating changes within each component. Updates to the Model (e.g., data structure changes) do not require modifications to the View or Controller unless the user interface or business logic changes.  
  
**5. Generics and Collections:**

**Advantages:**

**Type Safety:** Generics ensure compile-time type checking, reducing the likelihood of runtime errors such as ClassCastException. This ensures that only the intended types of objects are added to collections.

**Code Reusability:** Generics allow classes and methods to be written in a way that can handle multiple types, promoting code reuse without sacrificing type safety.

**Performance:** Generics in Java use type erasure, meaning that generic type information is removed after compilation. This improves performance by eliminating the need for runtime type checks and casts.

**Readability and Maintainability:** Using generics makes code more readable and understandable by explicitly specifying the types of objects being manipulated. This enhances code maintainability as it becomes easier to understand the intent behind the code.

**How Generics Helped Implement Collections in My System:**1. The CompositeFilter class uses generics to manage a collection of different filtering strategies. This allows the class to handle various types of filters uniformly without sacrificing type safety.public interface Filtering {  
List<Hotel> filter(List<Hotel> hotels);  
}  
import java.util.ArrayList;  
import java.util.List;  
public class CompositeFilter implements Filtering {  
private List<Filtering> filters = new ArrayList<>();  
public void addFilter(Filtering filter) {  
filters.add(filter);  
}  
@Override  
public List<Hotel> filter(List<Hotel> hotels) {  
List<Hotel> filteredHotels = hotels;  
for (Filtering filter : filters) {  
filteredHotels = filter.filter(filteredHotels);  
}  
return filteredHotels;  
}}  
In this example, the use of a generic List<Filtering> ensures that only objects implementing the Filtering interface can be added to the list. This prevents runtime errors and enforces type safety, making the code more robust and easier to maintain.

2. The SystemBooking class uses generics to filter a list of hotels based on various criteria. This method demonstrates the power of generics in handling collections of specific types.  
  
public List<Hotel> filterHotels(List<Hotel> hotels, List<String> filter, List<String> amenitiesHotel, List<String> amenitiesRoom, Date start, Date finish, double priceMin,

double priceMax, int rating, String location) {

boolean b = false;

this.filter = new CompositeFilter();

for (String f : filter) {

switch (f.toLowerCase()) {

case "amenities to hotel":

if (amenitiesHotel != null) {

Filtering filtering = new FilterByAmenitiesHotel(amenitiesHotel);

this.filter.addFilter(filtering);

b = true;

break;

}

case "amenities to room":

if (amenitiesRoom != null) {

Filtering filtering = new FilterByAmenitiesRoom(amenitiesRoom);

this.filter.addFilter(filtering);

b = true;

break;

}

case "date":

if (start != null && finish != null) {

Filtering filtering = new FilterByDate(start, finish);

this.filter.addFilter(filtering);

b = true;

break;

}

case "price":

if (priceMin > 0 && priceMax >= priceMin) {

Filtering filtering = new FilterByPrice(priceMin, priceMax);

this.filter.addFilter(filtering);

b = true;

break;

}

case "rating":

if (rating > 0) {

Filtering filtering = new FilterByRating(rating);

this.filter.addFilter(filtering);

b = true;

break;

}

case "location":

if (location != null) {

Filtering filtering = new FilterByLocation(location);

this.filter.addFilter(filtering);

b = true;

break;

}

default:

break;

}

}

if (b) {

return this.filter.filter(hotels);

}

return null;

}

**6. Exception Handling Strategies:**In my project, the exception-handling strategy involves catching and handling exceptions to ensure that the application can gracefully handle unexpected situations without crashing.   
I manage handling exceptions to ensure that it wont crash by:   
validation checks, try – catch blocks,going with debugging for validations.

**Effective exception handling contributes to robust application development by:**

**Preventing Crashes:** By catching exceptions, the application can handle errors gracefully and continue running.  
**Providing Meaningful Error Messages:** Users can understand what went wrong and take corrective actions.  
**Ensuring Data Integrity:** Proper exception handling ensures that the application’s state remains consistent even when errors occur.  
**Enhancing Debugging and Maintenance:** Clear and specific exception messages help developers diagnose and fix issues more efficiently.

**Example from my code:** public Person findPersonById(long personId) {//person  
 if (personId < 0) {  
 return null;  
 }  
 try {  
 Person person = this.persons.get(personId);  
 switch (person) {  
 case Manager manager -> {  
 return manager;  
 }  
 case User user -> {  
 return user;  
 }  
 default -> {  
 return null;  
 }  
 }  
 } catch (NullPointerException e) {  
 return null;  
 }  
 }

This was crucial because if it didn’t find the person then it throws exception of null pointer exception and the program will crash.

**7. Code Optimization and Efficiency:  
Efficient Data Structures:**Choosing the right data structures is crucial for optimizing performance. In the hotel booking system, appropriate data structures were selected based on the specific requirements of each component.  
**Lists and Maps:**ArrayList was used for collections that require fast random access and iteration.  
HashMap was used for key-value pairs to ensure average O(1) time complexity for insertions, deletions and lookups.  
**Optimized Algorithms:**Algorithms were carefully designed and optimized to ensure they run efficiently, particularly for operations involving large datasets.  
**Filtering and Sorting:**  
Efficient filtering strategies were implemented using the Stream API for concise and parallel processing.  
Sorting was optimized using the Stream API's sorted method, which leverages efficient sorting algorithms. **Assessing Trade-offs Between Code Readability, Ease of Maintenance, and Performance:**In optimizing code for performance and efficiency, it's essential to balance several factors, including readability, maintainability, and performance. Here's how these trade-offs are assessed in the context of the hotel booking system:  
**Code Readability:  
Importance**: Readable code is easier to understand, review, and debug. It facilitates collaboration among team members and aids in onboarding new developers.  
**Trade-offs**: Highly optimized code can sometimes become less readable. For instance, using complex algorithms or low-level optimizations might obfuscate the logic.  
**Approach**: To maintain readability, the hotel booking system uses clear and descriptive variable names, comments, and well-defined methods. For example, the filtering logic is encapsulated in specific classes implementing the Filtering interface, making the code modular and understandable.  
  
**Ease of Maintenance:  
Importance**: Maintainable code allows for easier updates, bug fixes, and feature enhancements. It ensures the long-term sustainability of the application. **Trade-offs**: Optimization techniques like inlining functions or using advanced data structures can complicate maintenance.  
**Approach**: The system uses design patterns such as Strategy for filtering and sorting to encapsulate variations in behavior. This makes it easier to modify or extend functionality without affecting other parts of the system.

**Performance:  
Importance**: Performance is critical for providing a responsive user experience, especially in data-intensive applications like hotel booking systems.  
**Trade-offs**: Highly readable and maintainable code can sometimes be less efficient. For example, using higher-level abstractions or standard library methods might be slower than hand-optimized code.  
**Approach**: Performance-critical sections of the code, such as filtering and sorting, are optimized using appropriate data structures and algorithms. Where necessary, parallel processing is employed to speed up operations without compromising readability and maintainability.

**8. Testing and Debugging:**

My approach to testing is to think first and Anticipate where code might fail and address those points proactively.  
By checking Input Validation, I'm preventing invalid data from causing crashes or unexpected behavior.  
Incremental Testing, verify the correctness of the code at each step of development.  
My approach to debugging is by using print statements, step by step debugging and refining structure.  
  
**The Challenges I face with are:**   
**Handling incorrect input:** The initial design did not adequately account for all types of incorrect input, leading to crashes and unexpected behavior.  
**Structuring the code correctly:** The initial structure of the code was not optimal, leading to difficulties in managing and testing my system.  
**Effective debugging:** Identifying and fixing bugs in a complex system with multiple interacting components.  
  
**How did I overcome those challenges:**   
**Handling incorrect input:** Implemented comprehensive input validation to catch and handle invalid data early. This included checks for null values, date ranges, and other constraints.  
**Structuring the code correctly:** Refactored the code to follow best practices and design patterns. This involved reorganizing classes and methods to improve readability and maintainability.  
**Effective debugging:** Used a combination of print statements and IDE debugging tools to trace the execution flow and inspect variable states. This helped in pinpointing the exact location and cause of bugs.  
  
**How do design patterns affect the testing process:**  
The design patterns contribute significantly to the modularity and testability of the system, enabling comprehensive testing and easier maintenance. They help in building a robust structure that facilitates the detection and resolution of issues early in the development process.  
Also it helps by impacting the testing process by enhancing modularity, promoting code reuse, and simplifying complex systems.