Here are **four design patterns** from your list that are likely implemented or can be implemented in your **restaurant booking system** and how they apply to your current project. I’ve also identified the specific areas in your project where you’re applying or could apply these patterns, so you can add comments to explain them:

**1. Singleton Pattern (Creational)**

* **Where in your project**:
  + You've already used the Singleton pattern in your BookingService class by annotating it with @Service. This tells Spring to manage it as a singleton, ensuring that there is only one instance of the BookingService throughout the application.
* **Why it fits**:
  + This pattern ensures that the BookingService has a single shared state across the entire application, maintaining consistency.

**Where to add a comment**:

@Service

public class BookingService {

// Singleton pattern: Ensures that only one instance of this service is created and used

}

**2. Factory Pattern (Creational)**

* **Where in your project**:
  + You can apply the **Factory pattern** to your Table creation logic. In your code, you have a loop for generating multiple Table objects with programmatically determined attributes like table IDs and sizes.
  + Though it's lightweight usage, it still encapsulates the creation logic, which is the essence of the Factory pattern.
* **Why it fits**:
  + This pattern allows you to centralize the logic for creating objects, which is helpful when object creation logic becomes complex or needs to be reused across your system.

**Where to add a comment**:

// Factory pattern: Creating tables in a loop to centralize the object creation logic and avoid duplication.

for (int i = 0; i < 4; i++) {

tables.add(new Table(idCounter++, 4)); // Factory pattern encapsulates table creation

}

**3. Command Pattern (Behavioral)**

* **Where in your project**:
  + The **Command pattern** could be applied to the reservation actions like creating, deleting, and updating reservations. These actions could be encapsulated into **command objects** and then executed as needed. Although your project may not have fully implemented this yet, it's a logical place for the Command pattern.
* **Why it fits**:
  + This pattern allows you to decouple the request for an action (e.g., making a reservation) from the execution of that action. It can also be helpful if you want to add features like undo/redo or logging the commands executed.

**Where to add a comment**:

// Command pattern: Each action (create, update, delete) can be encapsulated as a command object for flexibility

public class CreateReservationCommand implements Command {

private BookingService bookingService;

private Reservation reservation;

public CreateReservationCommand(BookingService bookingService, Reservation reservation) {

this.bookingService = bookingService;

this.reservation = reservation;

}

@Override

public void execute() {

bookingService.createReservation(reservation); // Command execution

}

}

**4. Adapter Pattern (Structural)**

* **Where in your project**:
  + You can use the **Adapter pattern** to integrate the **MongoDB repository** (ReservationRepository) with your internal data model (Reservation). This helps to separate concerns by adapting different data sources or formats into your application's required format.
* **Why it fits**:
  + If you were to introduce another external data source (like a third-party API), you could easily adapt the data into your internal application model using this pattern.

**Where to add a comment**:

// Adapter pattern: Adapt external data model (MongoDocument) into internal application model (Reservation)

public Reservation toReservation(ReservationDocument doc) {

return new Reservation(doc.getId(), doc.getCustomerName(), doc.getTableNumber(), doc.getArrivalTime(), doc.getDepartureTime());

}

**Summary Table with Comments**

| **Pattern Name** | **Where in Your Project** | **Comment Example** |
| --- | --- | --- |
| **Singleton** | BookingService class | @Service ensures only one instance of BookingService exists. |
| **Factory** | Table creation logic in the service | Loops through table creation, centralizing creation logic. |
| **Command** | Reservation creation/deletion/update actions | Encapsulate actions (create, delete, update) as command objects. |
| **Adapter** | MongoDB repository to internal model mapping | Adapt MongoDB documents to internal Reservation objects. |

These patterns will help you maintain cleaner code, improve scalability, and make your application more modular. Let me know if you want more detailed examples of how to implement any of these!