Nombre del proyecto:

Desarrollo de Aplicación Usando POO y Patrones de Diseño

Curso:

Java Developer 20

Integrantes:

Valeria Michelle Barbero Rivera

Cristian Alexander Ramírez Juárez

Judith Esther Arévalo Guardado

Grupo N°7

Fecha de entrega:

Domingo 06 de Julio de 2025

Tabla de contenido

Introducción

El desarrollo de sistemas de software orientados a objetos exige no solo una correcta implementación funcional, sino también una arquitectura sólida que facilite el mantenimiento y la extensión del código. En este proyecto se implementa un sistema de reservas para un hotel utilizando Java, aplicando los principios de la programación orientada a objetos (POO) y patrones de diseño para resolver problemas comunes de forma estructurada y reutilizable.

El sistema permite gestionar habitaciones, clientes y servicios adicionales como spa o desayuno, combinando funcionalidades sin necesidad de modificar la estructura base del sistema. Cada decisión de diseño se tomó con el objetivo de lograr una aplicación flexible, modular y fácil de ampliar.

Este documento describe los principales patrones utilizados en el desarrollo del sistema: Singleton, Observer, Factory y Decorator. A través de ejemplos concretos de código y explicaciones detalladas, se evidencia cómo cada patrón contribuye a la organización, claridad y escalabilidad de la aplicación.

Objetivos

- Diseñar e implementar un sistema de reservas hoteleras que gestione de forma eficiente habitaciones, clientes y servicios adicionales, utilizando principios sólidos de programación orientada a objetos.
- Aplicar patrones de diseño adecuados para resolver necesidades específicas del sistema, promoviendo un código limpio, extensible y fácil de mantener.
- Documentar las decisiones de arquitectura y diseño, explicando de forma clara cómo cada patrón fue implementado y cuál es su aporte dentro de la estructura general del sistema.

Descripción del sistema

El sistema de reservas hoteleras permite registrar y gestionar las reservaciones realizadas por los clientes del hotel. Ofrece funcionalidades para crear, confirmar y cancelar reservas, así como verificar la disponibilidad de habitaciones. Además, permite añadir servicios adicionales como desayuno o spa, aplicando de manera flexible distintas combinaciones según las necesidades del cliente. Todo el sistema está construido en Java, con una arquitectura modular que incorpora patrones de diseño para asegurar su escalabilidad y facilidad de mantenimiento.

Patrones de diseño implementados

Patrón Singleton

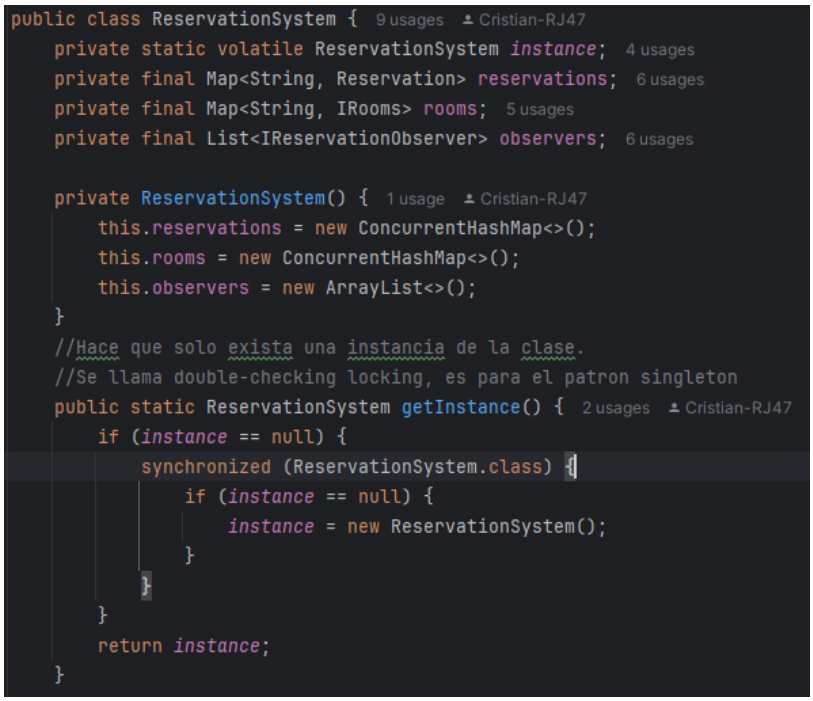
El patrón Singleton se utiliza para garantizar que una clase tenga una única instancia y proporcionar un punto global de acceso a ella. En sistemas donde es crucial que solo exista un único objeto manejando un recurso o estado compartido —como un sistema de reservas— el patrón Singleton es una solución estándar.

Su implementación

El sistema de reservas debe mantener un estado consistente que incluya todas las habitaciones, reservas y observadores. Permitir múltiples instancias podría generar inconsistencias o conflictos al acceder o modificar los datos. Por ello, limitar la creación a una única instancia facilita el control centralizado y evita condiciones de carrera en sistemas multihilo.

Se implementa usando la técnica double-check locking para asegurar que la instancia se cree una única vez y que el acceso a la misma sea seguro en entornos concurrentes.

- **Variable estática volatile:** La instancia instance se declara como private static volatile para garantizar la visibilidad inmediata del objeto a todos los hilos después de su creación.
- **Constructor privado:** El constructor de ReservationSystem es privado para impedir que otras clases creen nuevas instancias.
- **Método getInstance():** Este método provee la instancia única, verificando primero si ya existe sin sincronizar para optimizar el acceso. Solo si no existe, sincroniza el bloque donde se vuelve a verificar la instancia y se crea si es necesario.



```java
public class ReservationSystem {  9 usages  Cristian-RJ47
    private static volatile ReservationSystem instance;  4 usages
    private final Map<String, Reservation> reservations;  6 usages
    private final Map<String, IRooms> rooms;  5 usages
    private final List<IReservationObserver> observers;  6 usages

    private ReservationSystem() {  1 usage  Cristian-RJ47
        this.reservations = new ConcurrentHashMap<>();
        this.rooms = new ConcurrentHashMap<>();
        this.observers = new ArrayList<>();
    }
    //Hace que solo exista una instancia de la clase.
    //Se llama double-checking locking, es para el patron singleton
    public static ReservationSystem getInstance() {  2 usages  Cristian-RJ47
        if (instance == null) {
            synchronized (ReservationSystem.class) {
                if (instance == null) {
                    instance = new ReservationSystem();
                }
            }
        }
        return instance;
    }
```

Ventajas de su implementación

- Evita creación innecesaria de instancias.
- Garantiza que sólo una instancia exista durante toda la ejecución.
- Optimiza el acceso evitando sincronización después de la inicialización.
- Permite manejar correctamente el estado global de reservas, habitaciones y observadores.

Patrón Observer

El patrón Observer se utiliza para definir una relación de uno a muchos entre objetos, de manera que cuando un objeto cambia su estado, todos sus dependientes (observadores) son notificados automáticamente y pueden reaccionar ante ese cambio sin que el sujeto conozca detalles específicos de ellos. Esto permite un diseño desacoplado y extensible.

Su implementación

En ReservationSystem, el patrón Observer se implementa para manejar eventos relacionados con las reservas, como la creación, confirmación y cancelación de una reserva.

Esto permite que distintas acciones adicionales (como enviar un correo electrónico o registrar un log) puedan reaccionar a estos eventos sin que ReservationSystem tenga que gestionar esas acciones directamente, manteniendo la responsabilidad única y facilitando la extensión.

Interfaz Observador (IReservationObserver)

Define los métodos que cualquier observador debe implementar para reaccionar a eventos específicos relacionados con una reserva.

```java
public interface IReservationObserver {  11 usages  2 implem
    void reservationCreated(Reservation reservation);
    void reservationCancelled(Reservation reservation);
    void reservationConfirmed(Reservation reservation);
}
```

Sujeto (Observable) – ReservationSystem

Mantiene una lista de observadores y métodos para agregar o eliminar observadores. Cuando ocurre un evento relevante, notifica a todos los observadores llamando a los métodos correspondientes.

```java
public void addObserver(IReservationObserver observer) { observers.add(observer); }

public void removeObserver(IReservationObserver observer) { observers.remove(observer); }

private void notifyReservationCreated(Reservation reservation) {  1 usage  Cristian-RJ47
    for (IReservationObserver observer : observers) {
        observer.reservationCreated(reservation);
    }
}

private void notifyReservationCancelled(Reservation reservation) {  1 usage  Cristian-RJ47
    for (IReservationObserver observer : observers) {
        observer.reservationCancelled(reservation);
    }
}

private void notifyReservationConfirmed(Reservation reservation) {  1 usage  Cristian-RJ47
    for (IReservationObserver observer : observers) {
        observer.reservationConfirmed(reservation);
    }
}
```

Observadores Concretos

Ejemplos de observadores que reaccionan a los eventos son EmailNotifier y SystemLogger.

EmailNotifier envía correos electrónicos al cliente cuando una reserva cambia de estado.

```java
@Override  1 usage  Cristian-RJ47
public void reservationCreated(Reservation reservation) {
    sendEmail(reservation.getCustomer().getEmail(),
            subject: "Reservación creada. Id: " + reservation.getReservationId(),
            body: "Su reservación ha sido creada exitosamente.\n" + reservation.toString());
    System.out.println("✅ Email enviado: reservación creada para " + reservation.getCustomer().getName());
}
```

SystemLogger registra en consola logs con timestamp cuando ocurre un evento.

```java
public class SystemLogger implements IReservationObserver {  2 usages  Cristian-RJ47
    private static final DateTimeFormatter formatDataLog = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  1 usage

    @Override  1 usage  Cristian-RJ47
    public void reservationCreated(Reservation reservation) { loggerString( action: "RESERVATION_CREATED", reservation); }

    @Override  1 usage  Cristian-RJ47
    public void reservationCancelled(Reservation reservation) { loggerString( action: "RESERVATION_CANCELLED", reservation);

    @Override  1 usage  Cristian-RJ47
    public void reservationConfirmed(Reservation reservation) { loggerString( action: "RESERVATION_CONFIRMED", reservation);

    private void loggerString(String action, Reservation reservation) {  3 usages  Cristian-RJ47

        String timestamp = LocalDateTime.now().format(formatDataLog);

        String logEntry = "[" + timestamp + "] " + action +
                " - id reserva: " + reservation.getReservationId() +
                ", cliente: " + reservation.getCustomer().getName() +
                ", habitación: " + reservation.getRoom().getRoomNumber();

        System.out.println("📝 LOGGER: " + logEntry);
    }
}
```

Ventajas de su implementación

- Desacoplamiento: ReservationSystem no necesita saber qué acciones se ejecutan al cambiar una reserva, solo notifica.
- Extensibilidad: Se pueden agregar nuevos observadores (como notificaciones SMS, auditorías, etc.) sin modificar la clase central.
- Mantenimiento simplificado: Cambios en la forma de notificar o loguear se hacen en los observadores específicos, sin tocar la lógica de reservas.

Patrón Factory

El patrón Factory Method permite definir una interfaz para crear un objeto, pero delega la decisión de qué clase concreta instanciar a las subclases. Esto es útil cuando el sistema necesita crear objetos sin acoplarse directamente a sus clases específicas, lo que mejora la extensibilidad y el mantenimiento del código.

Su implementación

En este sistema, el patrón Factory Method se emplea para la creación de habitaciones del hotel, permitiendo que el sistema instancie habitaciones de distintos tipos (por ejemplo, Suite, HabitaciónSimple, etc.) sin depender directamente de la clase concreta.

Esto facilita la incorporación de nuevas categorías de habitaciones en el futuro sin necesidad de modificar el código de alto nivel, como el Main o el ReservationSystem.

Clase abstracta AbstractRoomsFactory

Define el método createRoom() que será implementado por las fábricas concretas para producir habitaciones específicas.

```java
public abstract class AbstractRoomsFactory {  1 usage  1 inheri
    public abstract IRooms createRoom(String roomNumber);
}
```

Fábrica concreta – SuiteFactory

Extiende AbstractRoomsFactory y retorna una instancia específica del tipo Suite.

```java
public class SuiteFactory extends AbstractRoomsFactory {
    @Override  2 usages  Cristian-RJ47
    public IRooms createRoom(String roomNumber) {
        return new Suite(roomNumber);
    }
}
```

Uso en Main.java

Desde la clase principal, se crea una instancia de la fábrica (SuiteFactory) y se llama al método createRoom() para obtener un objeto del tipo habitación sin necesidad de saber que es una Suite.

```java
// Creamos las habitaciones
SuiteFactory suiteFactory = new SuiteFactory();
IRooms suite1 = suiteFactory.createRoom( roomNumber: "S201");
IRooms suite2 = suiteFactory.createRoom( roomNumber: "S202");

system.addRoom(suite1);
system.addRoom(suite2);
```

Ventajas de su implementación

- Abstracción en la creación de objetos: El código cliente no necesita conocer los detalles de la implementación concreta de la habitación.
- Facilidad de extensión: Para agregar una nueva categoría de habitación, basta con crear una nueva clase y una nueva fábrica que la implemente.
- Mayor mantenibilidad: Se reduce el acoplamiento entre la lógica de negocio y las clases concretas, haciendo que el sistema sea más fácil de mantener.

Patrón Decorator

El patrón Decorator permite añadir responsabilidades o funcionalidades adicionales a un objeto de manera dinámica, sin alterar su estructura original ni modificar su clase base. Es ideal cuando se desea mantener la flexibilidad del sistema frente a combinaciones de funcionalidades que pueden variar.

Su implementación

En este proyecto, el patrón Decorator se utiliza para agregar servicios adicionales a las habitaciones, como desayuno (BreakfastServiceDecorator) o spa (SpaServiceDecorator).

Estos servicios pueden combinarse entre sí de forma flexible, sin necesidad de crear clases distintas para cada posible combinación.

Interfaz base de servicios: IService

```
public interface IService {  1
    String description();  4 us
    double price();  4 usages  4
}
```

Componente concreto: RoomService

Implementa la interfaz IService y representa el servicio básico asociado a una habitación.

```
public class RoomService implements IService {  2 usages  C
    private String roomNumber;  2 usages
    private double price;  2 usages

    public RoomService(String roomNumber, double price) {
        this.roomNumber = roomNumber;
        this.price = price;
    }

    @Override  4 usages  Cristian-RJ47
    public String description() {
        return "Servicio a la habitación " + roomNumber;
    }

    @Override  no usages  Cristian-RJ47
    public double price() {
        return price;
    }
}
```

Clase decoradora abstracta: ServiceDecorator

Implementa IService y contiene una referencia a otro IService. Es la base para todos los decoradores concretos.

```java
public abstract class ServiceDecorator implements IService {
    protected IService service;  5 usages

    public ServiceDecorator(IService service) {  2 usages  Cris
        this.service = service;
    }

    @Override  4 usages  2 overrides  Cristian-RJ47
    public String description() {
        return service.description();
    }

    @Override  4 usages  2 overrides  Cristian-RJ47
    public double price() {
        return service.price();
    }
}
```

Decoradores concretos (Desayuno y Spa)

Añaden funcionalidad sobre el servicio original sin modificarlo

```java
public class BreakfastServiceDecorator extends ServiceDecorator {

    public BreakfastServiceDecorator(IService roomService) {  1 usag
        super(roomService);
    }

    @Override  4 usages  Cristian-RJ47
    public String description() {
        return super.description() + "\n✅ servicio de desayuno";
    }

    @Override  4 usages  Cristian-RJ47
    public double price() {
        return super.price() + 30.0;
    }
}
```

```java
public class SpaServiceDecorator extends ServiceDecorator {  2 us
    public SpaServiceDecorator(IService service) {  1 usage  Crist
        super(service);
    }
    @Override  4 usages  Cristian-RJ47
    public String description() {
        return service.description() + "\n✅ servicio de Spa";
    }
    @Override  4 usages  Cristian-RJ47
    public double price() {
        return service.price() + 70.0;
    }
}
```

Uso en la aplicación Main.java

El decorador permite combinar servicios fácilmente al envolver un decorador sobre otro:

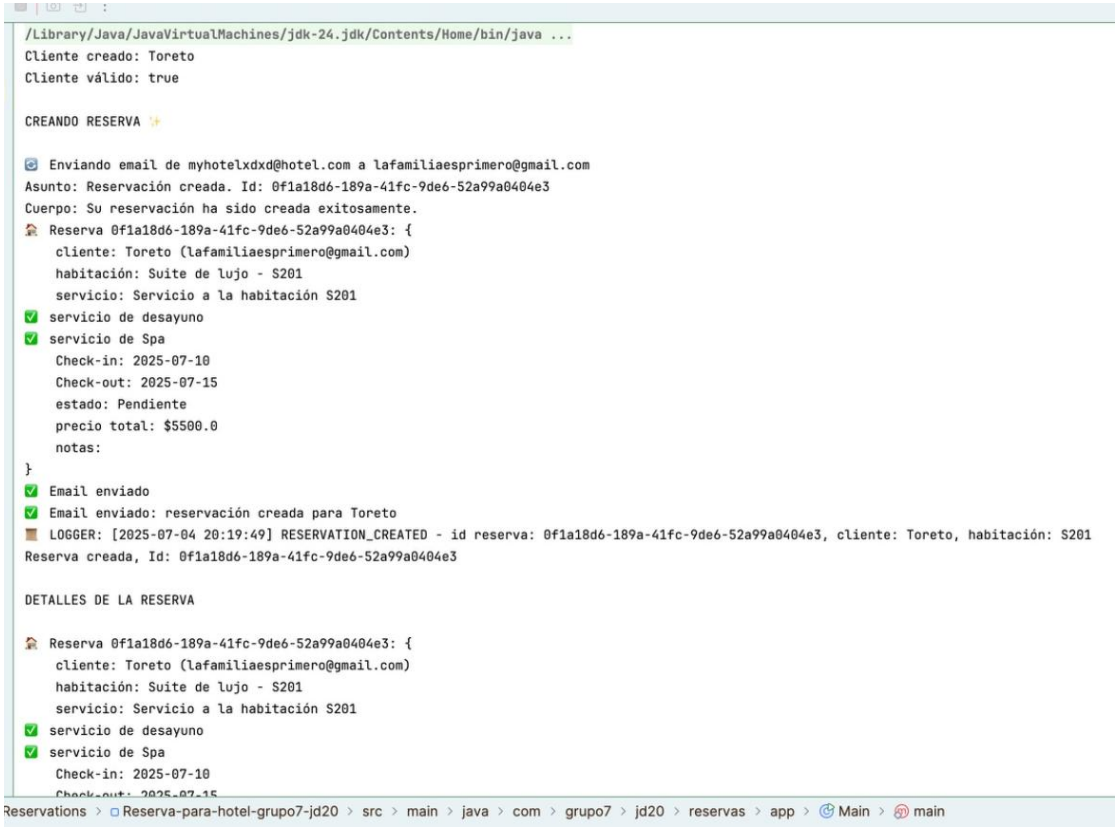
```
// Agregamos los servicios a la habitación seleccionada
IService serviceSuite1 = new RoomService(suite1.getRoomNumber(), suite1.getPrice());
serviceSuite1 = new SpaServiceDecorator(new BreakfastServiceDecorator(serviceSuite1));
```

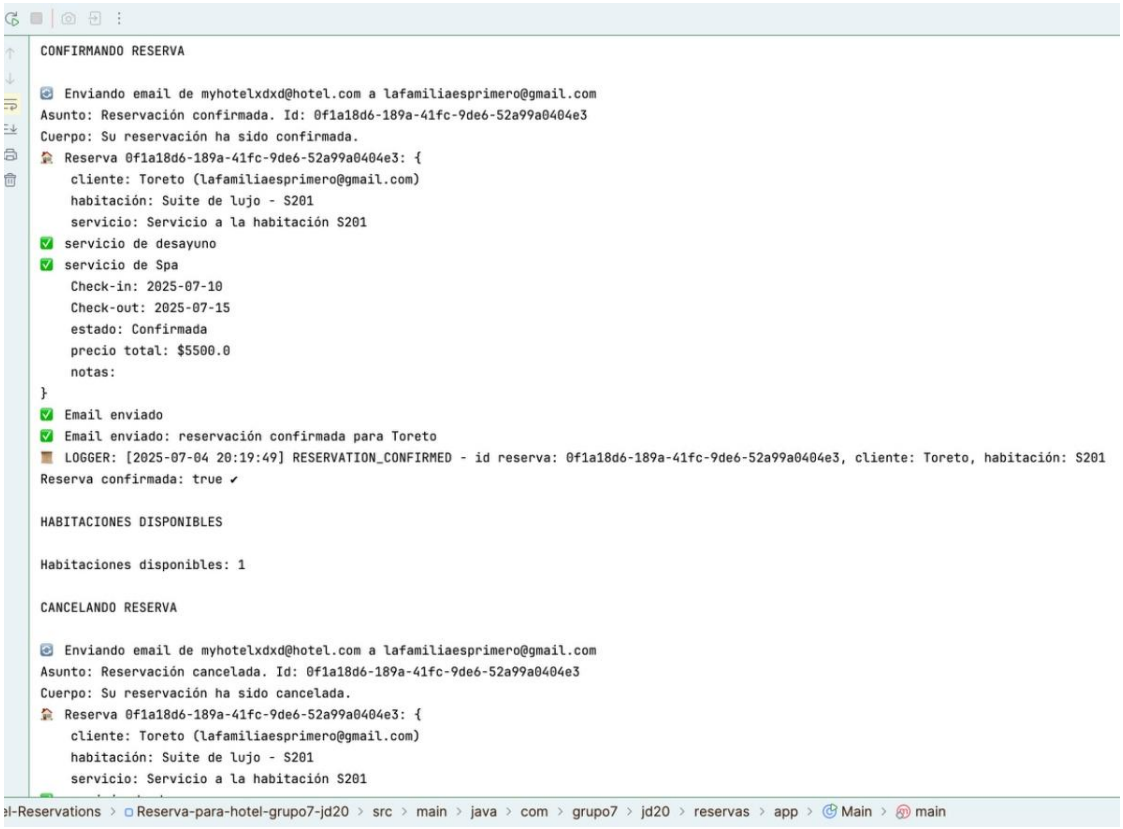
Esto crea un RoomService básico, al que se le agrega desayuno, y luego se le agrega spa. El resultado es un servicio que tiene todas esas características sin crear una clase nueva para "Suite con desayuno y spa".

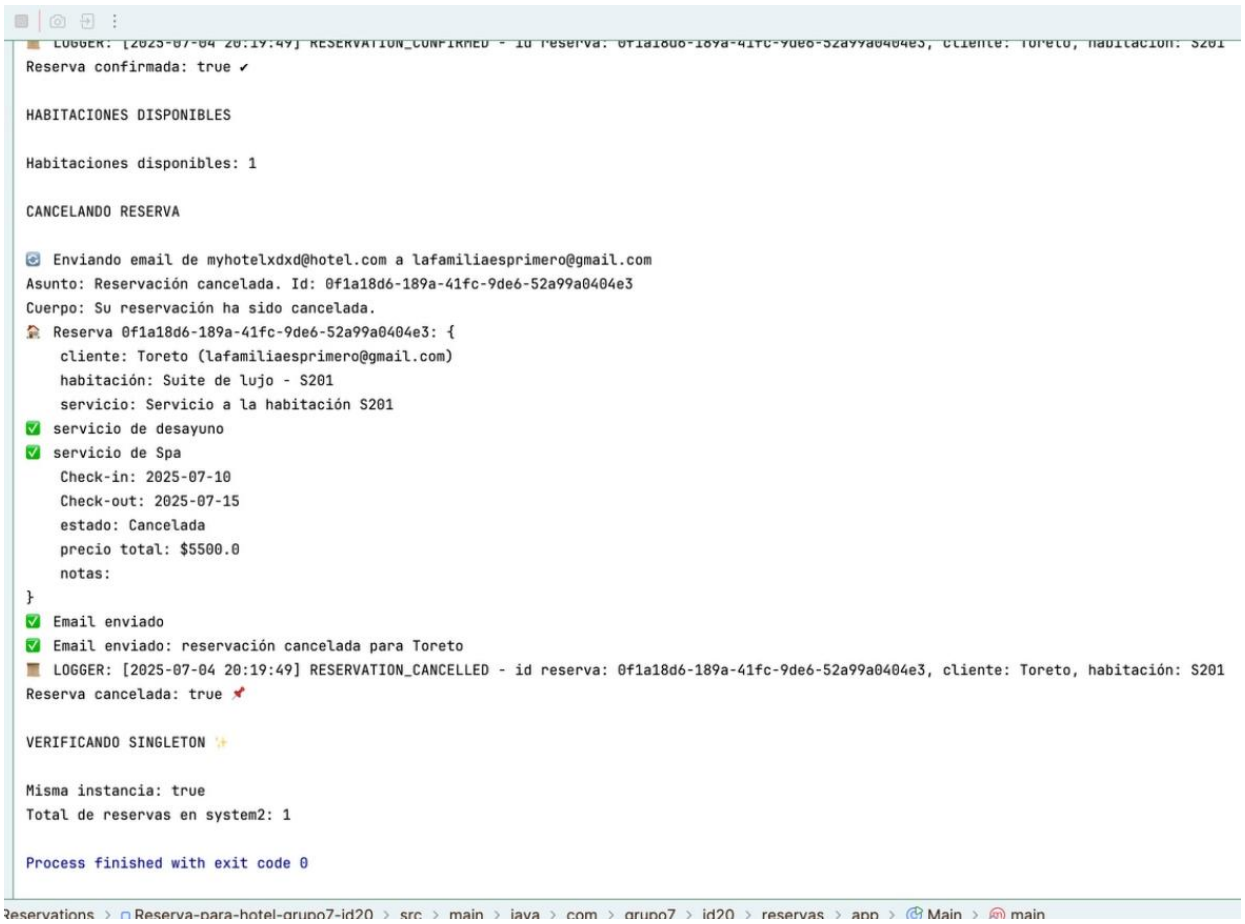
Ventajas de su implementación

- Flexibilidad para combinar servicios: Se pueden agregar múltiples funcionalidades sin necesidad de crear muchas subclases.
- Código más limpio: Evita la explosión de clases por cada posible combinación de servicios.
- Extensibilidad: Para agregar un nuevo servicio, basta con crear un nuevo decorador. No es necesario modificar el código existente.
- Principio de abierto/cerrado (OCP): Puedes extender el comportamiento sin modificar las clases originales.

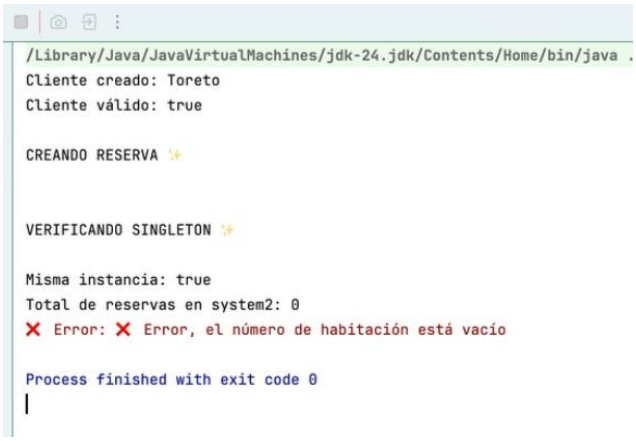
Ejecución del programa

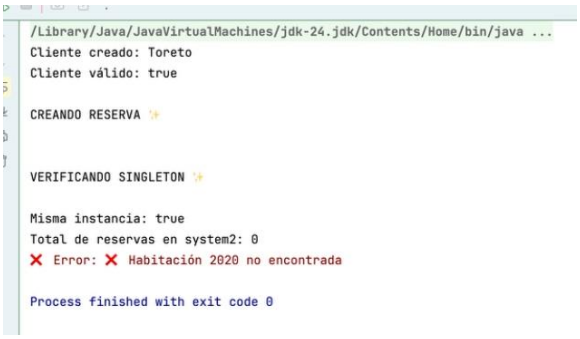


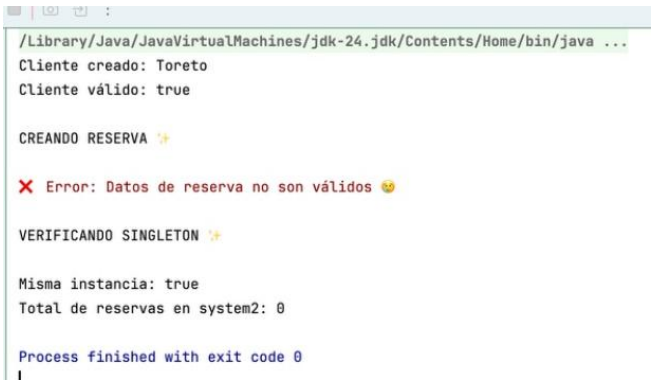

CONFIRMANDO RESERVA
Enviando email de myhotelxdxd@hotel.com a lafamiliaesprimero@gmail.com
Asunto: Reservación confirmada. Id: 0f1a18d6-189a-41fc-9de6-52a99a0404e3
Cuerpo: Su reservación ha sido confirmada.
Reserva 0f1a18d6-189a-41fc-9de6-52a99a0404e3: {
cliente: Toreto (lafamiliaesprimero@gmail.com)
habitación: Suite de lujo - S201
servicio: Servicio a la habitación S201
servicio de desayuno
servicio de Spa
Check-in: 2025-07-10
Check-out: 2025-07-15
estado: Confirmada
precio total: $5500.0
notas:
}
Email enviado
Email enviado: reservación confirmada para Toreto
LOGGER: [2025-07-04 20:19:49] RESERVATION_CONFIRMED - id reserva: 0f1a18d6-189a-41fc-9de6-52a99a0404e3, cliente: Toreto, habitación: S201
Reserva confirmada: true ✔
HABITACIONES DISPONIBLES
Habitaciones disponibles: 1
CANCELANDO RESERVA
Enviando email de myhotelxdxd@hotel.com a lafamiliaesprimero@gmail.com
Asunto: Reservación cancelada. Id: 0f1a18d6-189a-41fc-9de6-52a99a0404e3
Cuerpo: Su reservación ha sido cancelada.
Reserva 0f1a18d6-189a-41fc-9de6-52a99a0404e3: {
cliente: Toreto (lafamiliaesprimero@gmail.com)
habitación: Suite de lujo - S201
servicio: Servicio a la habitación S201
el-Reservations › Reserva-para-hotel-grupo7-jd20 › src › main › java › com › grupo7 › jd20 › reservas › app › Main › main


LOGGER: [2025-07-04 20:19:49] RESERVATION_CONFIRMED - id reserva: 0f1a18d6-189a-41fc-9de6-52a99a0404e3, cliente: Toreto, habitación: S201
Reserva confirmada: true ✔
HABITACIONES DISPONIBLES
Habitaciones disponibles: 1
CANCELANDO RESERVA
Enviando email de myhotelxdxd@hotel.com a lafamiliaesprimero@gmail.com
Asunto: Reservación cancelada. Id: 0f1a18d6-189a-41fc-9de6-52a99a0404e3
Cuerpo: Su reservación ha sido cancelada.
Reserva 0f1a18d6-189a-41fc-9de6-52a99a0404e3: {
cliente: Toreto (lafamiliaesprimero@gmail.com)
habitación: Suite de lujo - S201
servicio: Servicio a la habitación S201
servicio de desayuno
servicio de Spa
Check-in: 2025-07-10
Check-out: 2025-07-15
estado: Cancelada
precio total: $5500.0
notas:
}
Email enviado
Email enviado: reservación cancelada para Toreto
LOGGER: [2025-07-04 20:19:49] RESERVATION_CANCELLED - id reserva: 0f1a18d6-189a-41fc-9de6-52a99a0404e3, cliente: Toreto, habitación: S201
Reserva cancelada: true
VERIFICANDO SINGLETON
Misma instancia: true
Total de reservas en system2: 1
Process finished with exit code 0
Reservations › Reserva-para-hotel-grupo7-jd20 › src › main › java › com › grupo7 › jd20 › reservas › app › Main › main

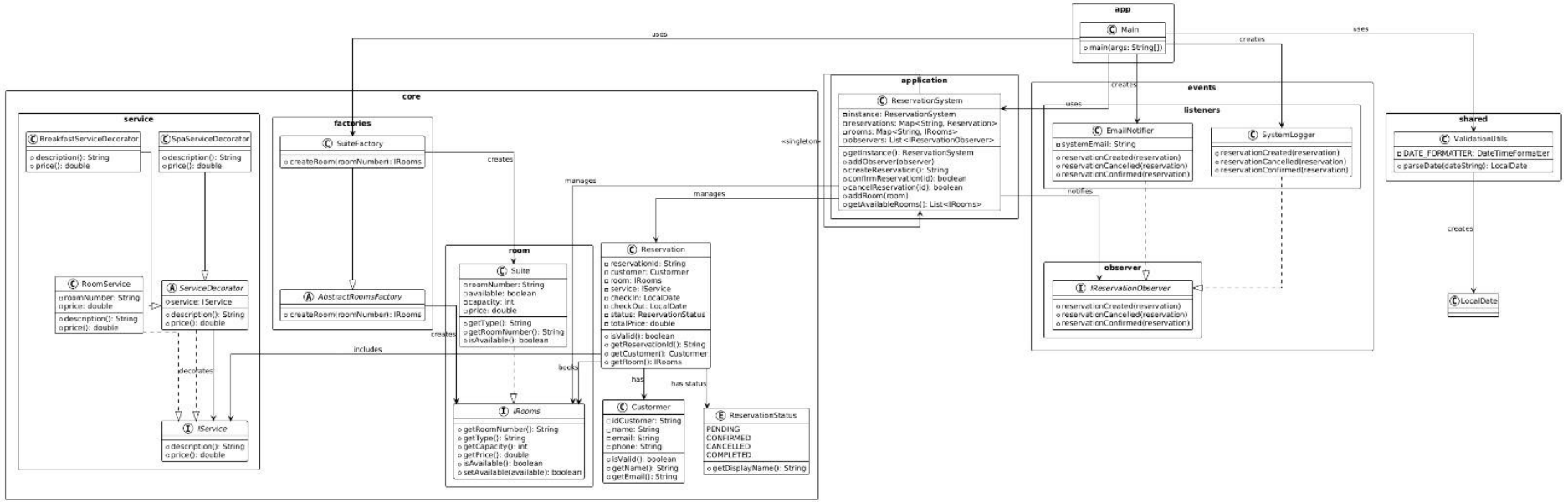
Manejo de errores y validaciones







UML


app
Main
main(args: String[])
uses
creates
core
service
BreakfastServiceDecorator
description(): String
price(): double
SpaServiceDecorator
description(): String
price(): double
RoomService
roomNumber: String
price: double
description(): String
price(): double
ServiceDecorator
service: IService
description(): String
price(): double
IService
description(): String
price(): double
decorates
factories
SuiteFactory
createRoom(roomNumber): IRooms
AbstractRoomsFactory
createRoom(roomNumber): IRooms
creates
includes
room
Suite
roomNumber: String
available: boolean
capacity: int
price: double
getType(): String
getRoomNumber(): String
isAvailable(): boolean
IRooms
getRoomNumber(): String
getType(): String
getCapacity(): int
getPrice(): double
isAvailable(): boolean
setAvailable(available): boolean
Reservation
reservationId: String
customer: Customer
room: IRooms
service: IService
checkIn: LocalDate
checkOut: LocalDate
status: ReservationStatus
totalPrice: double
isValid(): boolean
getReservationId(): String
getCustomer(): Customer
getRoom(): IRooms
books
has
has status
Customer
idCustomer: String
name: String
email: String
phone: String
isValid(): boolean
getName(): String
getEmail(): String
ReservationStatus
PENDING
CONFIRMED
CANCELLED
COMPLETED
getDisplayName(): String
«singleton»
manages
application
ReservationSystem
instance: ReservationSystem
reservations: Map<String, Reservation>
rooms: Map<String, IRooms>
observers: List<IReservationObserver>
getInstance(): ReservationSystem
addObserver(observer)
createReservation(): String
confirmReservation(id): boolean
cancelReservation(id): boolean
addRoom(room)
getAvailableRooms(): List<IRooms>
events
listeners
EmailNotifier
systemEmail: String
reservationCreated(reservation)
reservationCancelled(reservation)
reservationConfirmed(reservation)
SystemLogger
reservationCreated(reservation)
reservationCancelled(reservation)
reservationConfirmed(reservation)
notifies
observer
IReservationObserver
reservationCreated(reservation)
reservationCancelled(reservation)
reservationConfirmed(reservation)
shared
ValidationUtils
DATE_FORMATTER: DateTimeFormatter
parseDate(dateString): LocalDate
LocalDate