|  |
| --- |
| Integrantes |
| Edwin Romeo Rivas Díaz |
| Luis Ernesto Figueroa Vásquez |
| Kevin Alexander Aquino Vásquez |
| Luis Alexis Velázquez Godoy |
| Guillermo Alberto Asensio Jiménez |
| Moisés Roberto Hernández Hernández |
| Jonathan Alexander Ramírez Vázquez |

Se analizo cada clase y se aplicó cada principio y cómo implementarlo en el código.

Refactorización del Código:

1. Single Responsibility Principle (SRP)

Cada clase debe tener una única razón para cambiar. Al revisar las clases del código, encontramos que algunas clases están manejando múltiples responsabilidades, por ejemplo, la clase ReservationSystem, que maneja tanto la lógica de reservas como la de agregar clientes y habitaciones.

Refactorización:

* La clase ReservationSystem se divide en varias clases más pequeñas, como CustomerManager, RoomManager, y ReservationManager.
* ReservationSystem solo orquestará la interacción entre los diferentes gestores de entidades (clientes, habitaciones, reservas).

***// Clase ReservationManager***

***public class ReservationManager {***

***private final CustomerManager customerManager;***

***private final RoomManager roomManager;***

***public ReservationManager(CustomerManager customerManager, RoomManager roomManager) {***

***this.customerManager = customerManager;***

***this.roomManager = roomManager;***

***}***

***public void reserveRoom(Customer customer, Room room) {***

***// Lógica para hacer una reserva***

***System.out.println("Reserva realizada para " + customer.getName() + " en la habitación " + room.getRoomNumber());***

***}***

***}***

**2. Open/Closed Principle (OCP)**

Las clases deben estar abiertas para extensión, pero cerradas para modificación. Esto significa que el código debería poder extenderse sin necesidad de modificar las clases existentes.

**Refactorización:**

* Creamos una interfaz PaymentMethod y diferentes implementaciones de métodos de pago como CreditCardPayment y PayPalPayment. Esto permite agregar nuevos métodos de pago sin modificar las clases existentes.

***// Interfaz PaymentMethod***

***public interface PaymentMethod {***

***void processPayment(double amount);***

***}***

***// Implementación del método de pago con tarjeta de crédito***

***public class CreditCardPayment implements PaymentMethod {***

***@Override***

***public void processPayment(double amount) {***

***System.out.println("Procesando pago con tarjeta de crédito de la cantidad: " + amount);***

***}***

***}***

***// Implementación del método de pago con PayPal***

***public class PayPalPayment implements PaymentMethod {***

***@Override***

***public void processPayment(double amount) {***

***System.out.println("Procesando pago con PayPal de la cantidad: " + amount);***

***}***

***}***

**3. Liskov Substitution Principle (LSP)**

Los subtipos deben poder sustituir a sus supertipos sin afectar el comportamiento esperado. En este caso, las clases que representan diferentes tipos de habitaciones deben ser intercambiables sin afectar el flujo de trabajo.

**Refactorización:**

* Aseguramos que todas las clases que implementan la clase Room sean intercambiables. Si creamos una nueva clase SuiteRoom o SingleRoom, debe comportarse de la misma manera que cualquier otro Room dentro del sistema.

***// Clase base Room***

***public abstract class Room {***

***protected String roomNumber;***

***protected double price;***

***public abstract double getPrice();***

***}***

***// Implementación de SuiteRoom***

***public class SuiteRoom extends Room {***

***@Override***

***public double getPrice() {***

***return 150.0; // Precio específico para una Suite***

***}***

***}***

***// Implementación de SingleRoom***

***public class SingleRoom extends Room {***

***@Override***

***public double getPrice() {***

***return 75.0; // Precio específico para una habitación individual***

***}***

***}***

**4. Interface Segregation Principle (ISP)**

Las interfaces deben ser específicas y no forzar a las clases a implementar métodos que no utilizan.

**Refactorización:**

* En lugar de tener una única interfaz que abarque todas las funciones de un HotelService, dividimos la interfaz en diferentes partes según las necesidades de cada clase. Esto permite a las clases implementar solo los métodos que les son relevantes.

***// Interfaz de gestión de habitaciones***

***public interface RoomService {***

***void checkIn(Room room);***

***void checkOut(Room room);***

***}***

***// Interfaz de gestión de clientes***

***public interface CustomerService {***

***void addCustomer(Customer customer);***

***void removeCustomer(Customer customer);***

***}***

**5. Dependency Inversion Principle (DIP)**

Las clases deben depender de abstracciones, no de implementaciones concretas. En lugar de depender directamente de clases como Customer o Room, las dependencias se deben inyectar a través de interfaces.

**Refactorización:**

* Cambiar la dependencia directa entre clases por inyecciones de dependencias, utilizando interfaces y asegurándonos de que las clases no dependan directamente de implementaciones concretas.

***// Interfaz para la gestión de clientes***

***public interface ICustomerRepository {***

***void addCustomer(Customer customer);***

***void removeCustomer(Customer customer);***

***}***

***// Implementación concreta***

***public class CustomerRepository implements ICustomerRepository {***

***@Override***

***public void addCustomer(Customer customer) {***

***// Lógica para agregar el cliente***

***}***

***@Override***

***public void removeCustomer(Customer customer) {***

***// Lógica para eliminar el cliente***

***}***

***}***

**Resumido del código refactorizado:**

* **Responsabilidades separadas** en clases dedicadas como CustomerManager, RoomManager, ReservationManager.
* **Uso de interfaces** para permitir la extensión de funcionalidades como diferentes métodos de pago sin modificar el código existente.
* **Inyección de dependencias** para reducir el acoplamiento entre clases.

**Conclusión:**

Con esta refactorización hemos aplicado los principios SOLID al proyecto, haciendo que el código sea más mantenible, escalable y fácil de entender. Al seguir estos principios, el código está preparado para futuras extensiones sin necesidad de grandes modificaciones en la estructura existente. Las dependencias se gestionan a través de interfaces, las clases son más coherentes con el SRP y las extensiones se pueden añadir sin romper la funcionalidad existente gracias al OCP.

**Pruebas:**

**Texto

Descripción generada automáticamente**

**Texto

Descripción generada automáticamente**

Texto

Descripción generada automáticamente