Inyección de Dependencias

La Inyección de Dependencias es un patrón de diseño de software que promueve la separación de responsabilidades y reduce el acoplamiento entre componentes de una aplicación. En lugar de que una clase se encargue de crear o gestionar sus propias dependencias (por ejemplo, servicios o repositorios), estas dependencias son proporcionadas desde el exterior, típicamente a través de un constructor, un método o un campo.

Propósitos y Beneficios de la Inyección de Dependencias:

- **Desacoplamiento**: Permite que los componentes de un sistema se conecten entre sí a través de interfaces, en lugar de depender directamente de implementaciones concretas. Esto facilita el cambio de implementaciones sin afectar el resto del sistema.
- **Facilidad de Pruebas**: Facilita la sustitución de implementaciones reales por versiones simuladas (mocks) en pruebas unitarias, lo que mejora la capacidad para probar componentes de forma aislada.
- **Modularidad y Escalabilidad**: Permite desarrollar y mantener diferentes partes del sistema de manera independiente, y añadir nuevas funcionalidades con un impacto mínimo en el sistema existente.

Descripción del Programa

El siguiente programa es una aplicación simple para gestionar las reservas en un hotel. Utiliza la inyección de dependencias para separar las responsabilidades entre el procesamiento de pagos, el almacenamiento de reservas y la lógica de reservas. Esto mejora la flexibilidad y modularidad del sistema.

Componentes del Programa

1. **Interfaces**: Definen contratos para los servicios de pago y el repositorio de reservas, asegurando que cualquier implementación concreta cumpla con estos contratos.
2. **Implementaciones**: Proporcionan la lógica concreta para procesar pagos y almacenar reservas. Estas clases concretas implementan las interfaces definidas.
3. **Servicio de Reservas**: Utiliza las interfaces para manejar el proceso de reserva. Dependiendo de las interfaces inyectadas, puede cambiar la implementación concreta sin modificar la lógica de negocio.
4. **Configuración Manual**: El punto de entrada del programa configura manualmente las dependencias e invoca el servicio de reservas, demostrando cómo se inyectan las dependencias.

Código del Programa

1. **Interfaces**

   PagoService y HotelRepository definen los contratos para las operaciones de pago y almacenamiento de reservas. Estas interfaces permiten que cualquier implementación

específica siga el mismo contrato, lo que facilita la interoperabilidad y la flexibilidad del sistema.

PagoService.java

```
// Interfaz para el servicio de pagos
public interface PagoService {
   void procesarPago(double monto);
}
```

HotelRepository.java

```
// Interfaz para el repositorio de hotel
public interface HotelRepository {
   void guardarReserva(Reserva reserva);
}
```

2. **Implementaciones**

PagoServiceImpl implementa el procesamiento de pagos y HotelRepositoryImpl se encarga de almacenar reservas. Estas implementaciones concretas proporcionan la lógica real que se utiliza en el sistema.

PagoServiceImpl.java

```
// Implementación de PagoService
public class PagoServiceImpl implements PagoService {
   @Override
   public void procesarPago(double monto) {
      // Lógica de procesamiento de pago
      System.out.println("Pago de $" + monto + " procesado.");
   }
}
```

HotelRepositoryImpl.java

```
// Implementación de HotelRepository
public class HotelRepositoryImpl implements HotelRepository {
```

```
    @Override
    public void guardarReserva(Reserva reserva) {
        // Lógica para guardar la reserva
        System.out.println("Reserva guardada: " + reserva);
    }
}
```

3. **Clase de Reserva**

   Reserva es una clase simple que contiene información sobre una reserva en el hotel. Incluye el tipo de habitación y el monto, así como métodos para acceder a estos datos y para su representación textual.

   Reserva.java

```
// Clase para representar una reserva
public class Reserva {
    private String tipoHabitacion;
    private double monto;

    public Reserva(String tipoHabitacion, double monto) {
        this.tipoHabitacion = tipoHabitacion;
        this.monto = monto;
    }

    public String getTipoHabitacion() {
        return tipoHabitacion;
    }

    public double getMonto() {
        return monto;
    }
```

```
    @Override
    public String toString() {
        return "Reserva [tipoHabitacion=" + tipoHabitacion + ", monto=" + monto + "]";
    }
}
```

4. **Servicio de Reservas**

ReservaService maneja el proceso de reserva. Recibe las dependencias PagoService y HotelRepository a través del constructor, lo que permite que el servicio se mantenga desacoplado de las implementaciones concretas. Esta separación facilita la modificación o sustitución de las dependencias sin alterar la lógica del servicio de reservas.

ReservaService.java

```
// Servicio para manejar reservas
public class ReservaService {
    private final PagoService pagoService;
    private final HotelRepository hotelRepository;

    // Inyección de dependencias a través del constructor
    public ReservaService(PagoService pagoService, HotelRepository hotelRepository) {
        this.pagoService = pagoService;
        this.hotelRepository = hotelRepository;
    }

    public void realizarReserva(Reserva reserva) {
        // Procesar el pago
        pagoService.procesarPago(reserva.getMonto());

        // Guardar la reserva en el repositorio
        hotelRepository.guardarReserva(reserva);
```

```
        System.out.println("Reserva realizada exitosamente.");
    }
}
```

5. **Configuración y Ejecución**

   En la clase Principal, se crean las instancias concretas de PagoService y HotelRepository, y se inyectan en ReservaService. Luego, se utiliza el ReservaService para realizar una reserva, demostrando cómo se integran las diferentes partes del sistema.

   Principal.java

```
public class Principal {
    public static void main(String[] args) {
        // Crear las implementaciones
        PagoService pagoService = new PagoServiceImpl();
        HotelRepository hotelRepository = new HotelRepositoryImpl();

        // Crear el servicio con inyección de dependencias manual
        ReservaService reservaService = new ReservaService(pagoService, hotelRepository);

        // Crear una reserva y procesarla
        Reserva reserva = new Reserva("Suite Deluxe", 200.00);
        reservaService.realizarReserva(reserva);
    }
}
```

Este enfoque de inyección de dependencias asegura que el sistema sea flexible y modular, permitiendo cambios y extensiones sin afectar el código existente, y facilita la prueba y el mantenimiento del sistema en general.