# Análisis de Principios SOLID y Patrones de Diseño

Principios SOLID

1. Single Responsibility Principle (SRP) - Principio de Responsabilidad Única  
   Este principio establece que cada clase debe tener una única razón para cambiar, es decir, una única responsabilidad.  
   Implementación en tu solución:
   * CustomerController: Su única responsabilidad es manejar las solicitudes HTTP relacionadas con los clientes.
   * CustomerServiceImpl: Encargado exclusivamente de la lógica de negocio relacionada con los clientes.
   * CustomerRepository: Solo maneja las interacciones con la base de datos para la entidad Customer.
2. Open/Closed Principle (OCP) - Principio Abierto/Cerrado  
   Este principio establece que las clases deben estar abiertas para su extensión, pero cerradas para su modificación.  
   Implementación en tu solución:
   * La interfaz CustomerService permite extender funcionalidades (por ejemplo, agregando nuevos métodos) sin modificar la implementación actual.
   * CustomerRepository extiende JpaRepository, aprovechando la funcionalidad base de Spring Data JPA sin modificarla.
3. Liskov Substitution Principle (LSP) - Principio de Sustitución de Liskov  
   Este principio establece que las clases derivadas deben ser sustituibles por sus clases base sin alterar el comportamiento del programa.  
   Implementación en tu solución:
   * CustomerServiceImpl implementa CustomerService, cumpliendo con el contrato definido en la interfaz. Cualquier cliente que use CustomerService puede utilizar CustomerServiceImpl sin cambios adicionales.
4. Interface Segregation Principle (ISP) - Principio de Segregación de Interfaces  
   Este principio sugiere que las interfaces deben ser específicas para cada caso de uso, en lugar de contener métodos innecesarios.  
   Implementación en tu solución:
   * CustomerService tiene una interfaz dedicada a las operaciones necesarias para los clientes. No incluye métodos irrelevantes o generales.
5. Dependency Inversion Principle (DIP) - Principio de Inversión de Dependencia  
   Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones.  
   Implementación en tu solución:
   * La clase CustomerController depende de la abstracción CustomerService, no de su implementación (CustomerServiceImpl). Esto facilita cambiar o extender la lógica de negocio sin afectar al controlador.

Patrones de Diseño

1. Inyección de Dependencias  
   Este patrón asegura que las dependencias de una clase sean proporcionadas externamente, mejorando la modularidad y facilidad de prueba.  
   Implementación en tu solución:
   * CustomerController utiliza la anotación @RequiredArgsConstructor para inyectar el servicio CustomerService.
   * En CustomerServiceImpl, el repositorio CustomerRepository es inyectado mediante @Autowired.
2. Repository Pattern  
   Este patrón abstrae las interacciones con la base de datos, proporcionando una interfaz uniforme para acceder a las entidades.  
   Implementación en tu solución:
   * La interfaz CustomerRepository extiende JpaRepository, lo que sigue el patrón Repository para desacoplar la lógica de acceso a datos de la lógica de negocio.
3. Service Layer Pattern  
   Este patrón organiza la lógica de negocio en una capa separada, proporcionando un puente entre el controlador y el repositorio.  
   Implementación en tu solución:
   * La clase CustomerServiceImpl actúa como una capa de servicio, separando la lógica de negocio del acceso a datos (CustomerRepository) y del controlador (CustomerController).
4. Factory Pattern (implícito en Spring)  
   Spring utiliza el patrón de fábrica para crear e inyectar instancias de beans, como el CustomerServiceImpl y CustomerRepository. Esto reduce la necesidad de inicializar manualmente los objetos.
5. Unit of Work (implícito en Spring Data JPA)  
   Spring Data JPA implementa este patrón al gestionar las transacciones y el contexto de persistencia, asegurando que todas las operaciones en la base de datos ocurran de manera consistente.

|  |  |  |
| --- | --- | --- |
| Principio/Patrón | Clase/Elemento Implementado | Descripción |
| SRP | CustomerController, CustomerServiceImpl, CustomerRepository | Cada clase tiene una única responsabilidad. |
| OCP | CustomerService | Se puede extender la funcionalidad sin modificar las clases existentes. |
| LSP | CustomerServiceImpl | Implementa CustomerService, cumpliendo su contrato. |
| ISP | CustomerService | Contiene solo métodos relevantes para los clientes. |
| DIP | CustomerController, CustomerService | Dependencia sobre abstracciones, no sobre implementaciones concretas. |
| Inyección de Dependencias | CustomerController, CustomerServiceImpl | Uso de @RequiredArgsConstructor y @Autowired. |
| Repository Pattern | CustomerRepository | Separa la lógica de acceso a datos. |
| Service Layer Pattern | CustomerServiceImpl | Implementa la lógica de negocio separada de la base de datos y el controlador. |
| Factory Pattern | Spring Framework | Crea instancias de beans automáticamente. |
| Unit of Work | Spring Data JPA | Manejo implícito de transacciones y contexto de persistencia. |