

CustomerMS

# **Principios SOLID y patrones de diseño**



YULY SANDRA CHOQUE RAMOS

Contenido

[**Análisis de principios SOLID y patrones de diseño** 0](#_Toc184142392)

[**Mejora Continua** 2](#_Toc184142393)

[SOLID y patrones de diseño 2](#_Toc184142394)

**Mejora Continua**

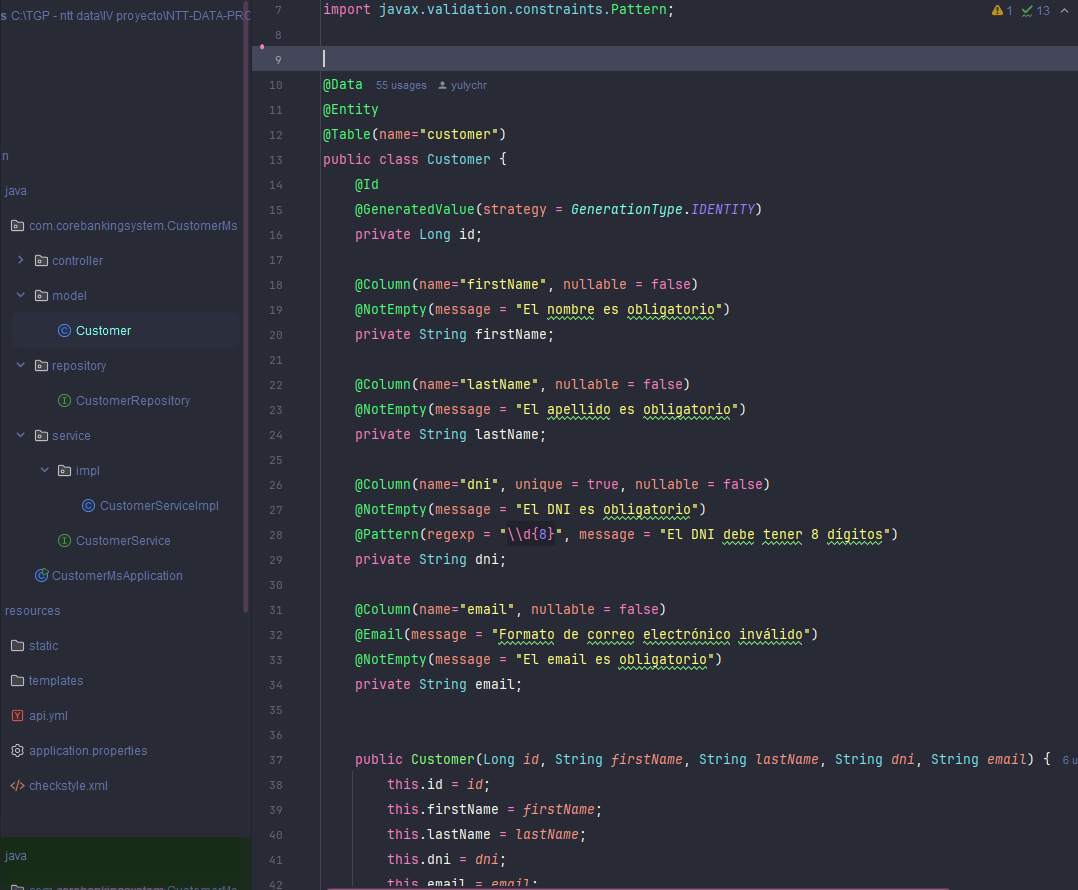
Este proyecto sigue un enfoque de **mejora continua**, lo que significa que las implementaciones son revisadas y refactorizadas regularmente para mejorar la calidad del código. Esto incluye mejorar la cobertura de pruebas, optimizar el diseño mediante la adherencia a los principios **SOLID** y refactorizar el código donde sea necesario para optimizar la estructura del código y el rendimiento.

## SOLID y patrones de diseño

En la clase Customer:

Principios SOLID

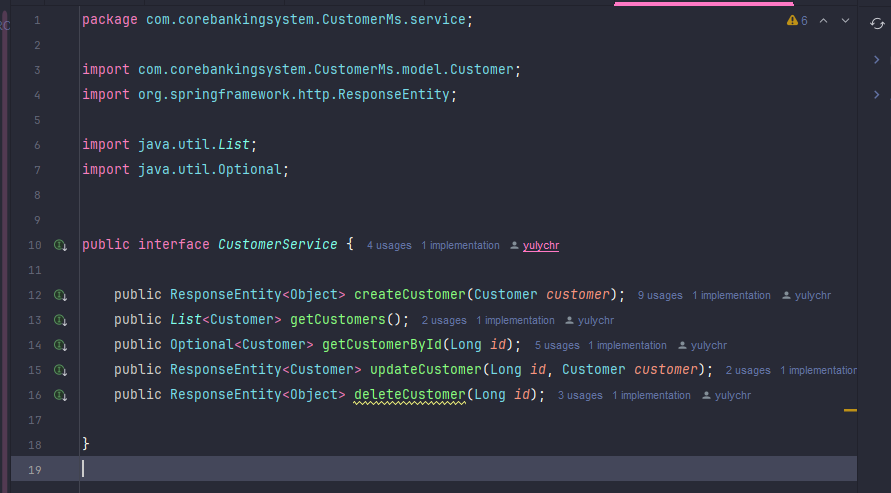
* SRP: Cumple bien con el principio, ya que la clase está enfocada en representar a un cliente.
* OCP: Cumple, aunque siempre puedes refactorizar para extender la clase sin modificarla.



En la interface CustomerService:

Principios SOLID

* SRP: La interfaz cumple con este principio ya que tiene una única responsabilidad de definir operaciones relacionadas con el servicio de clientes.
* OCP: Cumple con el principio, ya que puedes extender la interfaz sin modificarla.
* DIP: Cumple con este principio, ya que se está utilizando una abstracción (interfaz) en lugar de depender de implementaciones concretas.

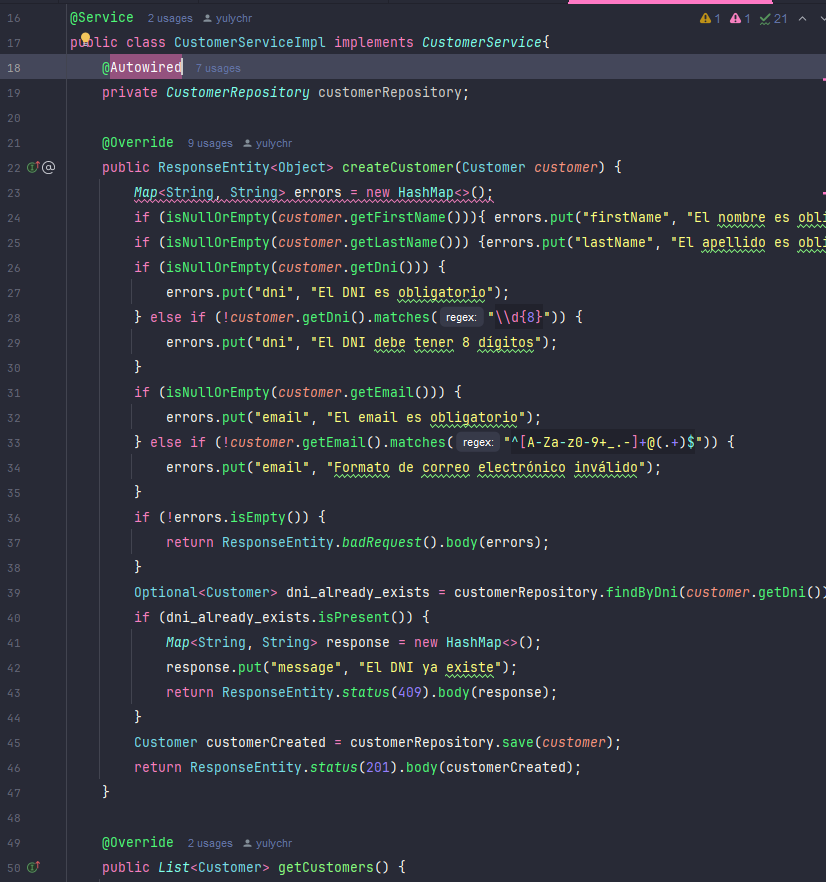


En la clase **CustomerServiceImpl**:

Principios SOLID

* OCP: Cumple, pero se puede hacer más extensible con más composición en lugar de herencia.
* ISP: Cumple, pero puede beneficiarse de dividir la interfaz en más pequeñas si la aplicación crece.
* DIP: Cumple, ya que la inyección de dependencias de Spring desacopla las clases.

Patrones de diseño: No cumple con ninguno, Aunque algunos patrones podrían aplicarse para mejorar el diseño (como Factory Method o Strategy), en su estado actual no se están utilizando explícitamente



En la clase **CustomerController:**

Principios SOLID

* SRP (Responsabilidad Única): El controlador sigue este principio, pero la lógica de negocio en deleteCustomer() podría moverse a un servicio.
* OCP (Abierto/Cerrado): Cumple, ya que puedes extender la funcionalidad sin modificar el controlador.
* LSP (Sustitución de Liskov): Cumple, ya que las subclases de CustomerService pueden sustituir la implementación sin problemas.
* DIP (Inversión de Dependencias): Cumple con el principio de inversión de dependencias, ya que el controlador depende de la interfaz y no de la implementación concreta.

