

AccountMS

# **Principios Solid y patrones de diseño**



YULY SANDRA CHOQUE RAMOS

Contenido

[**Principios Solid y patrones de diseño** 0](#_Toc184138365)

[**Mejora Continua** 2](#_Toc184138366)

[SOLID y patrones de diseño 2](#_Toc184138367)

**Mejora Continua**

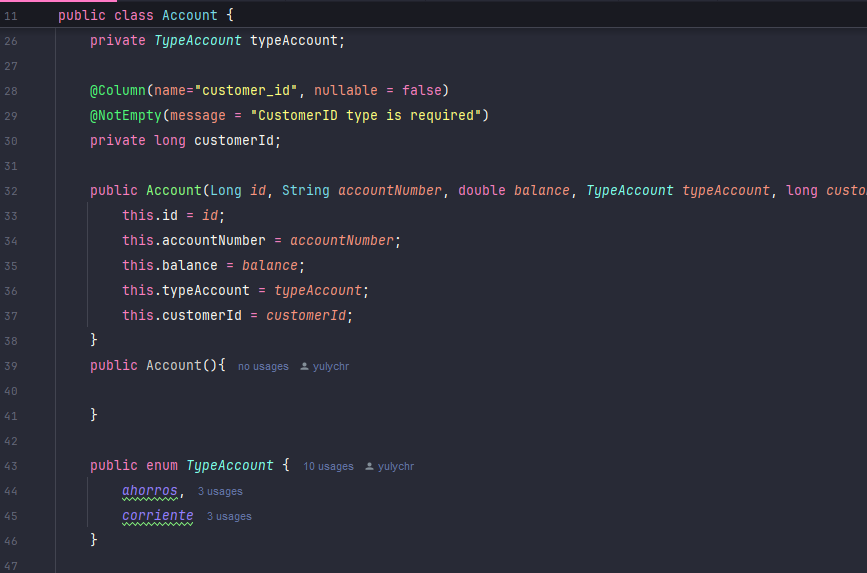
Este proyecto sigue un enfoque de **mejora continua**, lo que significa que las implementaciones son revisadas y refactorizadas regularmente para mejorar la calidad del código. Esto incluye mejorar la cobertura de pruebas, optimizar el diseño mediante la adherencia a los principios **SOLID** y refactorizar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

## SOLID y patrones de diseño

En la clase Account:

Principios SOLID

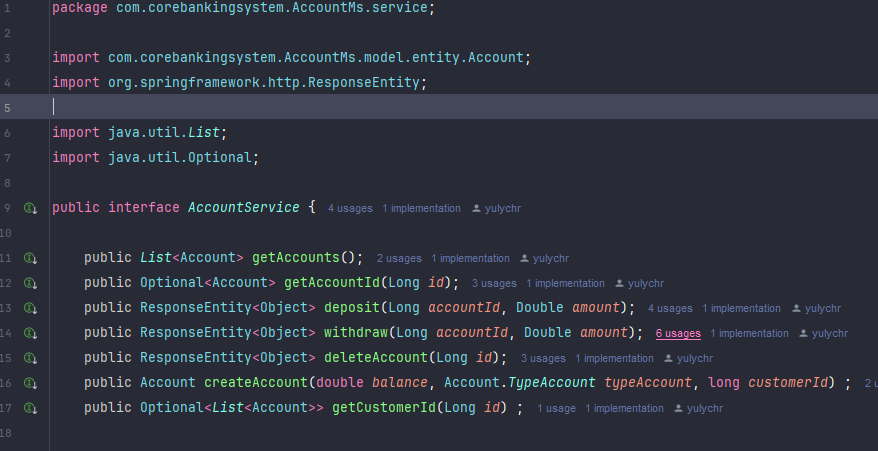
* SRP (Responsabilidad Única): Cumple, ya que Account se enfoca en representar la entidad de la cuenta bancaria y sus operaciones básicas.
* OCP (Abierto/Cerrado): Cumple, ya que se puede extender para agregar nuevos tipos de cuentas o nuevas operaciones sin modificar la clase base.
* DIP (Inversión de Dependencias): No se aplica directamente en la clase Account, pero se puede aplicar si hay servicios que interactúan con cuentas.



En la interface AccountService:

Principios SOLID

* SRP: La interfaz cumple con este principio ya que tiene una única responsabilidad de definir operaciones relacionadas con el servicio de cuentas.
* 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.



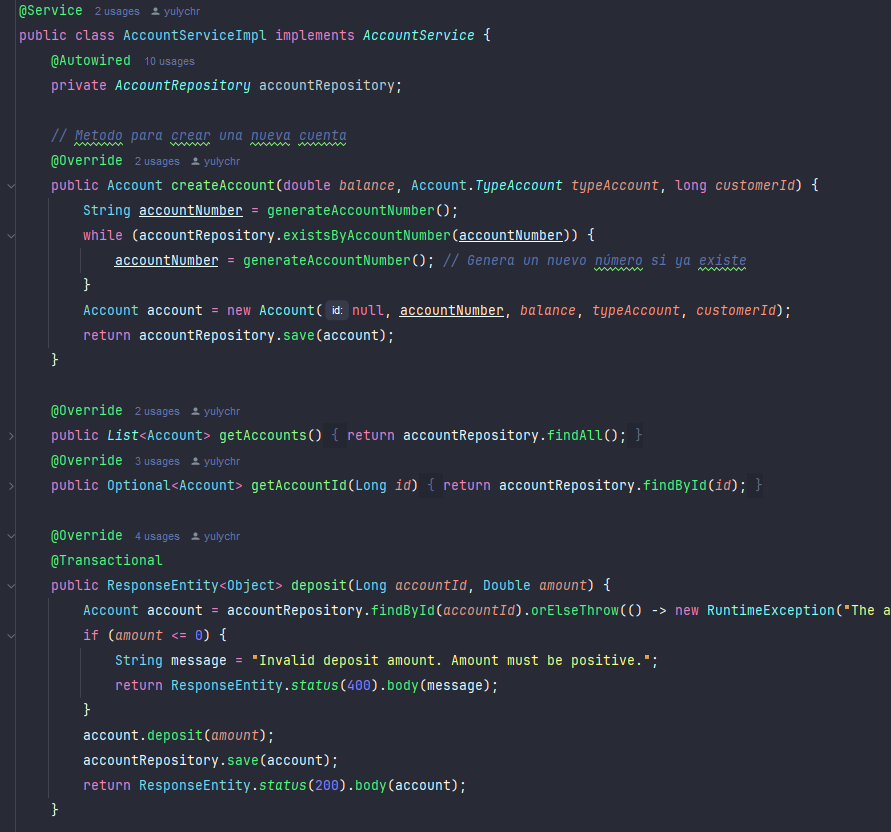
En la clase **AccountServiceImpl**:

Principios SOLID

* SRP: Cumple, AccountServiceImpl maneja exclusivamente la lógica de cuentas (crear, depositar, retirar, eliminar), con una única responsabilidad.
* OCP: Cumple, a clase es fácil de extender (se pueden agregar nuevos comportamientos sin modificar la clase actual).
* LPS: Cumple, la clase es extensible sin afectar el comportamiento actual, permitiendo herencia o implementación futura sin problemas.
* ISP: Cumple, la interfaz AccountService está centrada solo en métodos relevantes para cuentas, evitando métodos innecesarios.

Patrones de diseño:

* Factory Method: cumple parcialmente al tener un método para crear cuentas con una lógica interna, pero no está formalizado como un patrón.



En la clase **AccountController:**

Principios SOLID

* SRP (Responsabilidad Única): El controlador tiene la responsabilidad de manejar las solicitudes HTTP relacionadas con las cuentas, como la creación, consulta, actualización y eliminación.
* OCP (Abierto/Cerrado): Cumple, la clase está abierta a la extensión (se pueden agregar más endpoints fácilmente), pero no está completamente cerrada a la modificación.
* LSP (Sustitución de Liskov): No hay herencia explícita ni reemplazo de clases, ya que el controlador es una clase concreta que maneja las solicitudes.
* ISP: Cumple, el controlador no está haciendo cosas que no están relacionadas con su tarea principal.
* DIP (Inversión de Dependencias): El controlador depende de la abstracción (AccountService) y no de una implementación concreta. Esto permite que el servicio AccountService sea intercambiable, facilitando la prueba y la extensión del código.

Patrones de Diseño:

* Adapter: Cumple parcialmente. El patrón Adapter se usa para convertir una interfaz en otra, pero el uso del RestTemplate para realizar solicitudes a un servicio externo (para validar el customerId) podría considerarse una forma de adaptador.

