

TransactionMS

# Principios Solid y patrones de diseño



YULY SANDRA CHOQUE RAMOS

Contenido

[Principios Solid y patrones de diseño 0](#_Toc184152207)

[Mejora Continua 2](#_Toc184152208)

[SOLID y patrones de diseño 2](#_Toc184152209)

[Aplicación de Single Responsibility Principle a TransactionServiceImpl 2](#_Toc184152210)

[Aplicación de Single Open/Closed Principle a TransactionHandler 5](#_Toc184152211)

[Nota 8](#_Toc184152212)

[Aplicación de Interface Segregation Principle a TransactionOperation 8](#_Toc184152213)

[Aplicando Single Responsibility Principle a TransferOperationImpl, DepositOperationImpl y WithdrawalOperationImpl 10](#_Toc184152214)

[Resumen 13](#_Toc184152215)

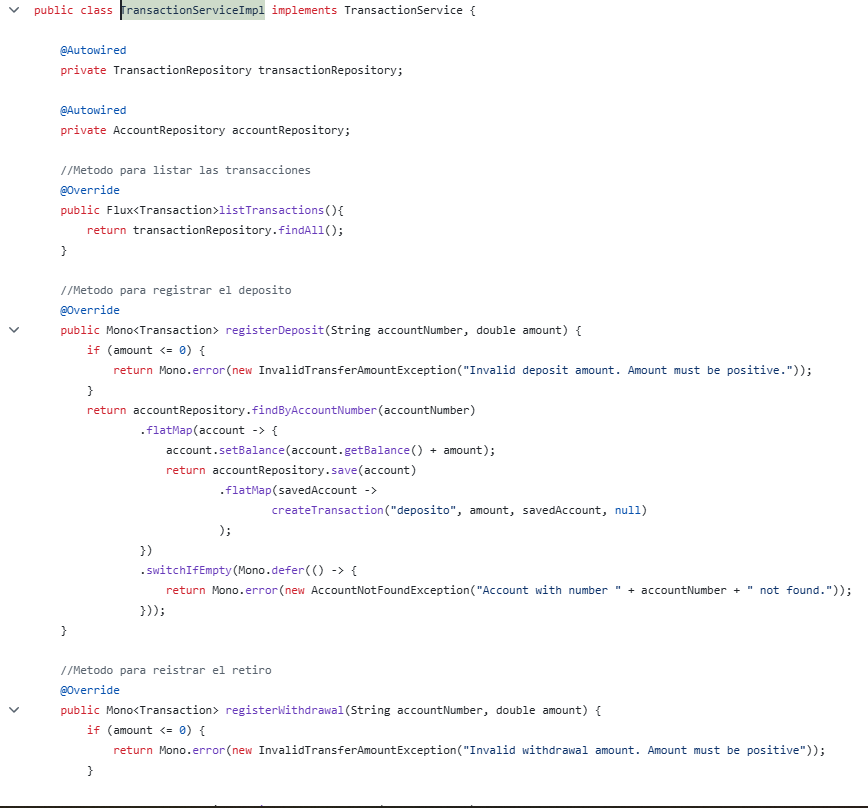
# Mejora Continua

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 patrones de diseño** para refactorizar el código donde sea necesario para optimizar la estructura del código y el rendimiento.

# SOLID y patrones de diseño

Se identificaron algunas mejoras de acuerdo a los patrones de diseño y principios SOLID, las cuales se implementaron de la siguiente manera.

## Aplicación de Single Responsibility Principle a TransactionServiceImpl



Se refactoriza **TransactionServiceImpl** para separar la lógica de las transacciones (depósito, retiro, transferencia) en clases más pequeñas, como **TransactionHandler,** **AccountUpdater** y **TransactionCreator**. Esto reducirá la complejidad del código y te permitirá aplicar las otras mejoras de manera más sencilla.

Cada una de estas nuevas clases tendrá una responsabilidad clara:

* **TransactionHandler**: Maneja la lógica de negocio para cada tipo de transacción (depósito, retiro, transferencia).
* **AccountUpdater**: Se encarga de la actualización del saldo de las cuentas involucradas en una transacción.
* **TransactionCreator**: Gestiona la creación y almacenamiento de las transacciones.

**commit:**

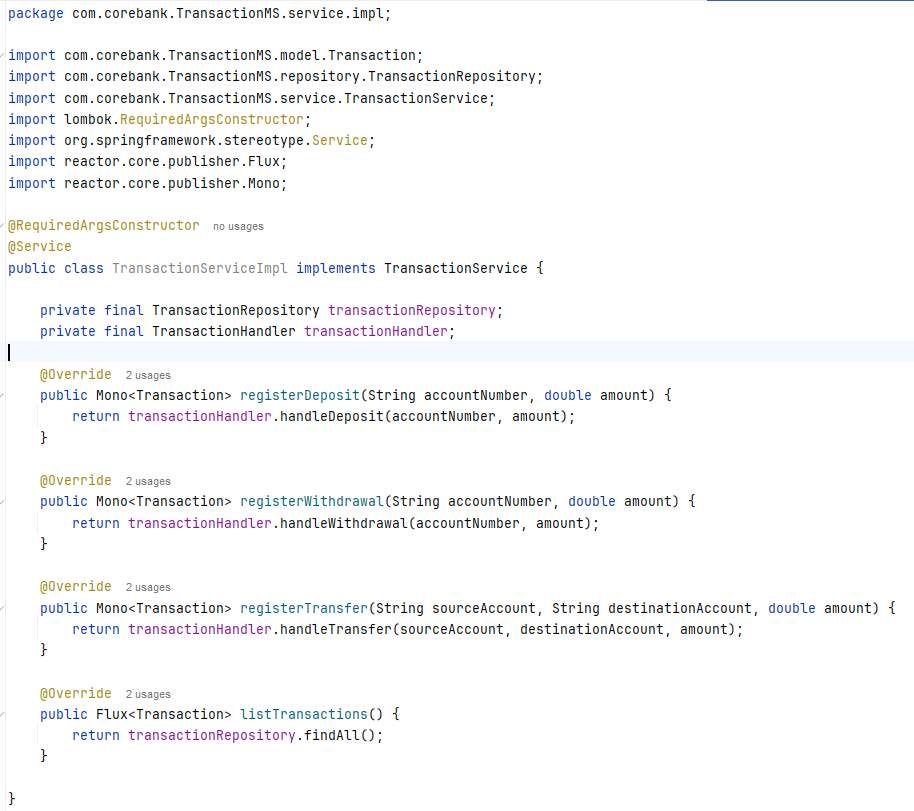
“refactor: Extract transaction logic into separate classes (TransactionHandler, AccountUpdater, TransactionCreator) to adhere to Single Responsibility Principle”

**Enlace:**

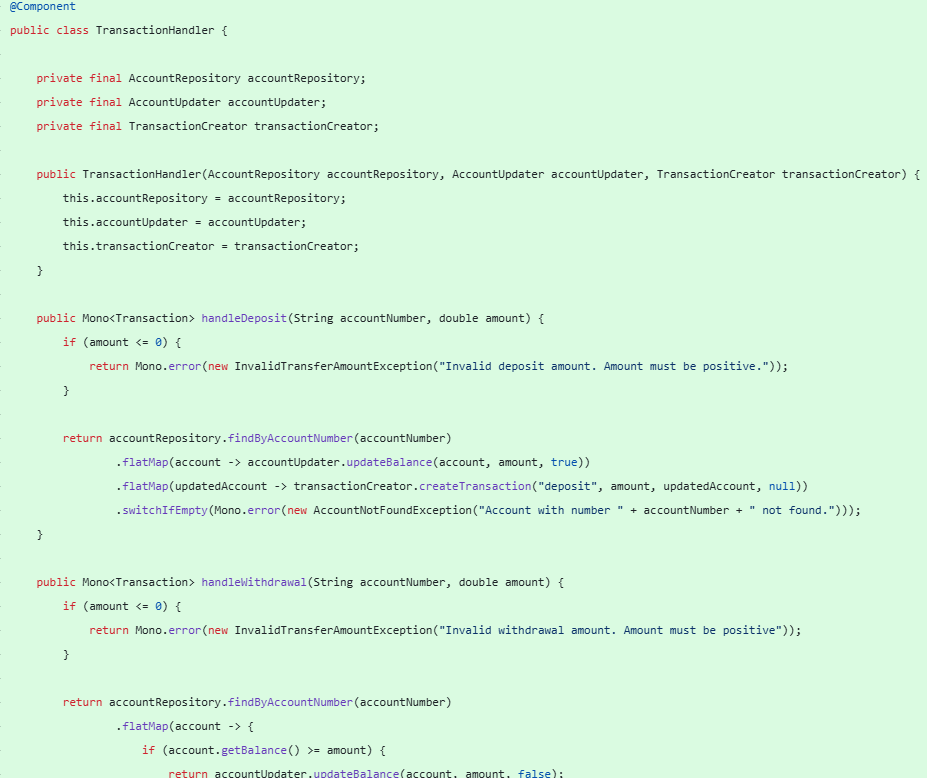
<https://github.com/yulychr/PF-NTT-DATA-TransactionMS/commit/18607e7c2d9bc7f368e50d57c437b33a19121b63>

Luego de la refactorización quedo del siguiente modo:

**TransactionServiceImpl**



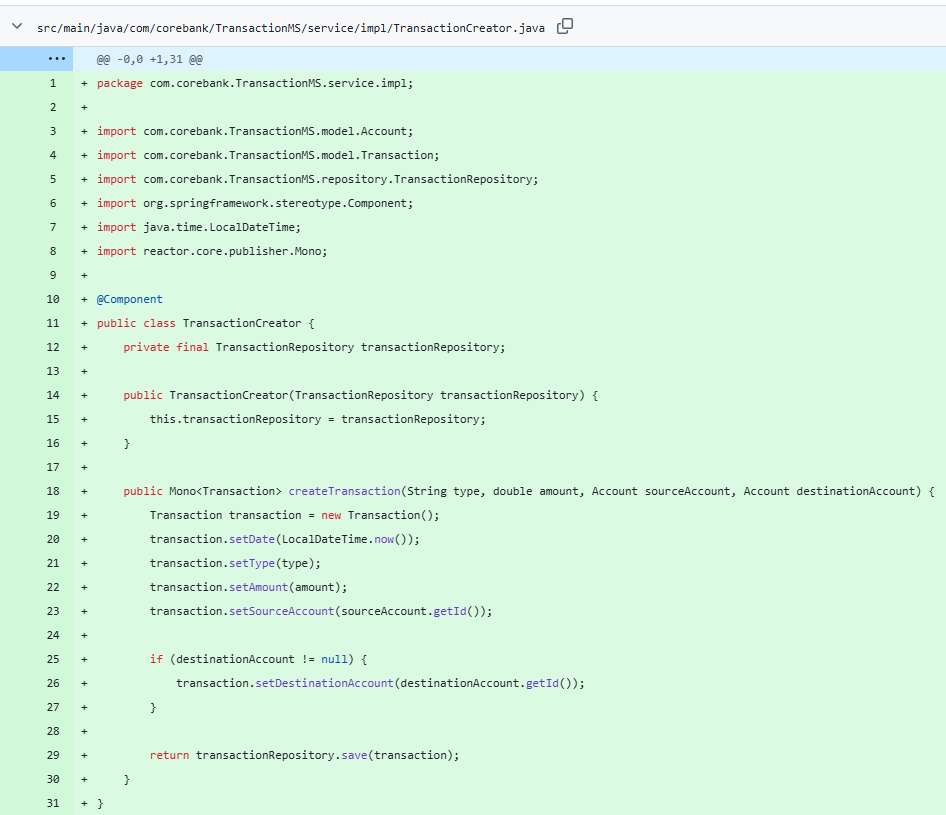
**TransactionHandler**



**AccountUpdater**



**TransactionCreator**



## Aplicación de Single Open/Closed Principle a TransactionHandler

El código está cerrado para modificaciones, ya que para agregar nuevas operaciones (como transferencia internacional, pagos, etc.), tendría que modificar **TransactionHandler**. Esto va en contra del principio de abierto/cerrado. Se puede usar el Patron de diseño Strategy, pero opte por nombrarlo …Operation para que vaya con la lógica del negocio.

Se abstrae las operaciones en una interfaz común y tener diferentes implementaciones de operaciones como **DepositOperation**, **WithdrawalOperation** y **TransferOperation,** así también se crea una Interfaz llamada **TransactionOperation** . Esto permitiría agregar nuevas operaciones sin modificar la clase TransactionHandler.

**commit:**

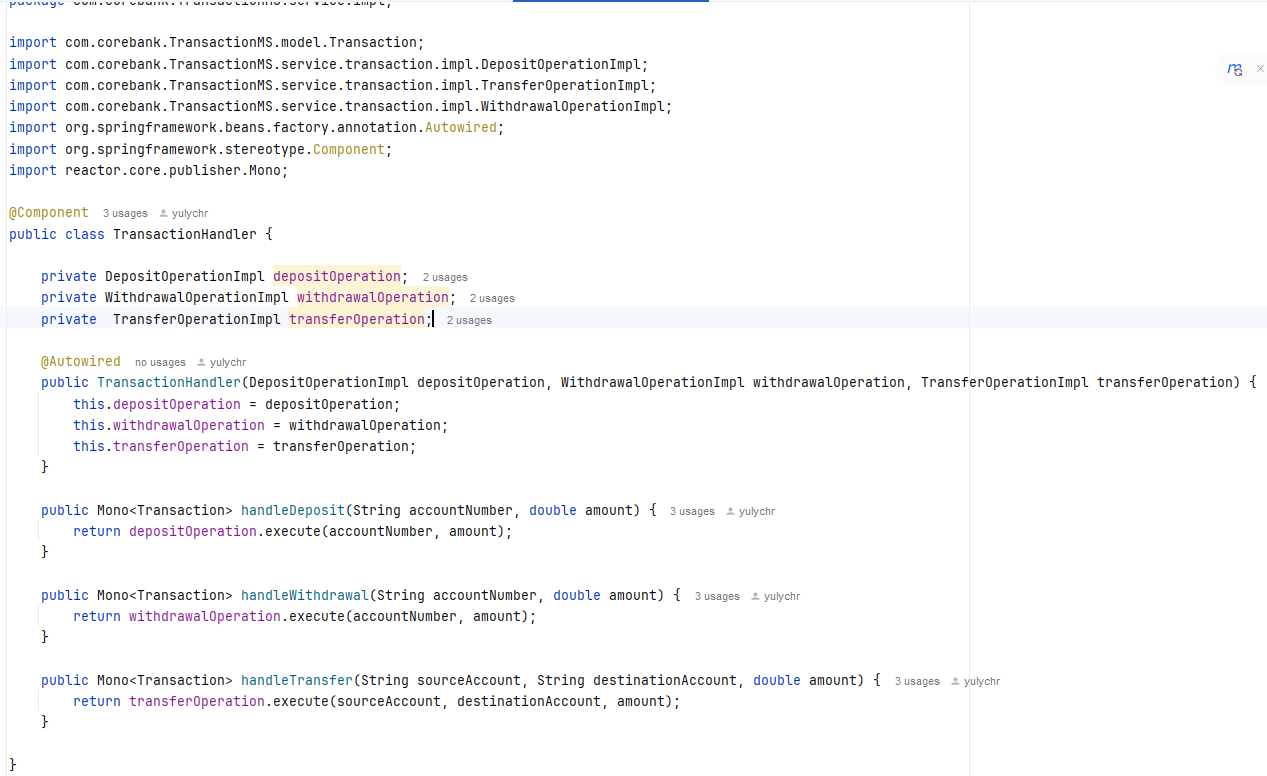
refactor: Exrtact transaction operations into separate classes (DepositOperation,WithdrawalOperation, TransferOperation) in service/transaction to adhere to Open/Closed Principle

**Enlace:**

<https://github.com/yulychr/PF-NTT-DATA-TransactionMS/commit/e11e2cd2fd3c4b89bb49993a3c279c07255acc2a>

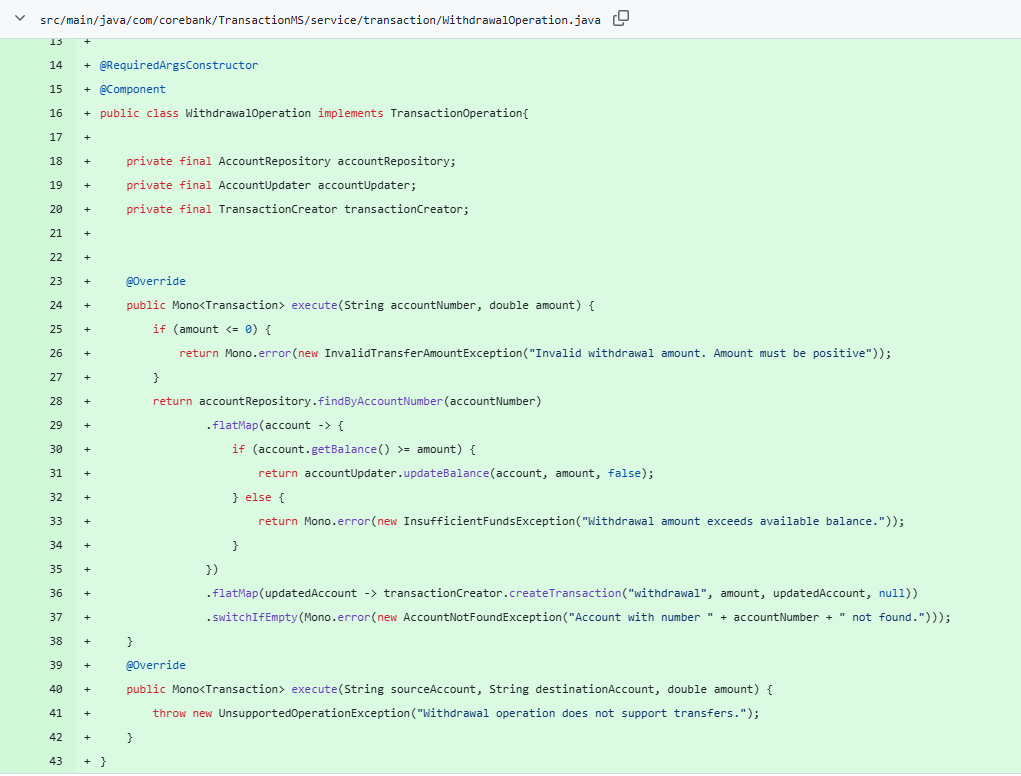
Luego de la refactorización quedo del siguiente modo:

**TransactionHandler**



**DepositOperation**

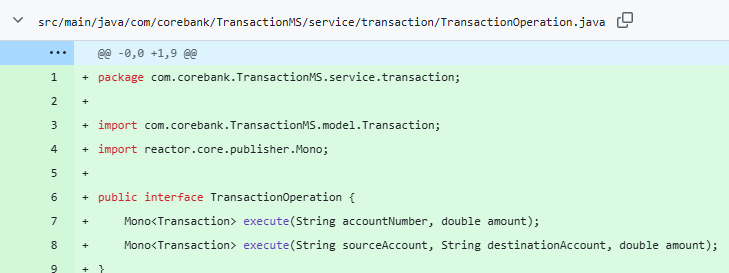
**WithdrawalOperation**



**TransferOperation**



**TransactionOperation**



# Nota

Se implementó WebClient para la comunicación con AccountMS para validar cuentas, procesar depósitos, retiros y transferencias utilizando APIs RESTful.

Se elimino y actualizo varios métodos.

Commit:

feat: Implement WebClient for communication with AccountMS to validate accounts, process deposits, withdrawals, and transfers using RESTful APIs

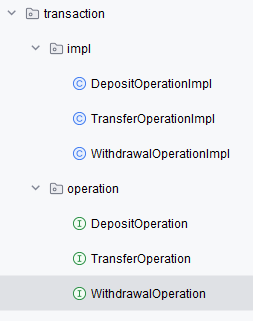
Enlace:

<https://github.com/yulychr/PF-NTT-DATA-TransactionMS/commit/c8816f5f28924bfb4dfe3e4c471a5cfe6eef7660>

## Aplicación de Interface Segregation Principle a TransactionOperation

Se analiza **TransactionOperation**  y actualmente, el método execute es genérico y no esta diseñado para ser extendido de manera modular. La interfaz **TransactionOperation** tiene métodos que no son útiles en todas las implementaciones (por ejemplo, el método execute que acepta solo un sourceAccount y amount no tiene sentido para una transferencia).

Por lo cual se divide la interfaz **TransactionOperation** en varias **interfaces** más pequeñas, como **DepositOperation**, **WithdrawalOperation** y **TransferOperation**.  
Por lo cual se pasa a renombrar las clases y se les aumenta …Impl



**commit:**

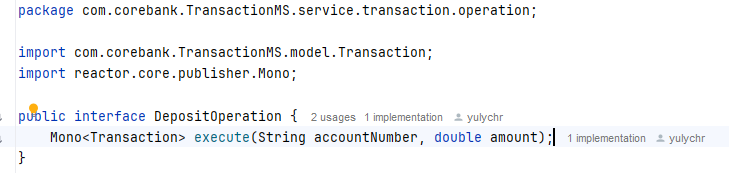
“refactor: Apply Interface Segregation Principle by splitting TransactionOperation into specialized interfaces (DepositOperation, WithdrawalOperation, TransferOperation)”

**Enlace:**

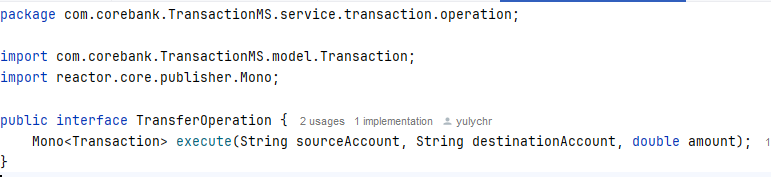
<https://github.com/yulychr/PF-NTT-DATA-TransactionMS/commit/ef29f38953f0cec1ee9506af84f90b19e891e056>

Luego de la refactorización quedo del siguiente modo:

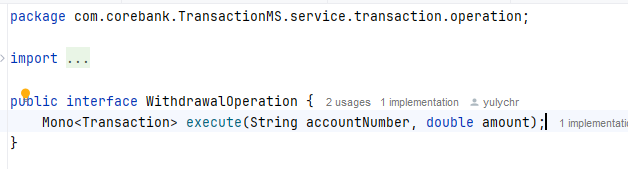
**Interface DepositOperation**



**Interface WithdrawalOperation**



**Interface TransferOperation**



## Aplicando Single Responsibility Principle a TransferOperationImpl, DepositOperationImpl y WithdrawalOperationImpl

Las clases: **TransferOperationImpl**, **DepositOperationImpl** y **WithdrawalOperatioImpl**, tienen demasiadas responsabilidades, gestionan las transferencias entre cuentas y están encargadas de varios aspectos como la validación de las cuentas, la verificación de fondos y la creación de transacciones.

Por lo cual se divide en varias clases, cada una encargada de una responsabilidad única.

* **AccountServiceT**: Encargada de validar la existencia de cuentas y actualizar saldos.
* **TransactionServiceT**: Encargada de crear transacciones.

**commit:**

refactor: Apply Single Responsibility Principle by splitting TransferOperation, DepositOperation, and WithdrawalOperation into specialized classes (AccountServiceT, TransactionServiceT)

**Enlace:**

<https://github.com/yulychr/PF-NTT-DATA-TransactionMS/commit/06dc09f9046a3b1cfcdcf73163240d6a7af25541>

Luego de la refactorización quedo del siguiente modo:

**TransferOperationImpl**



**DepositOperationImpl**



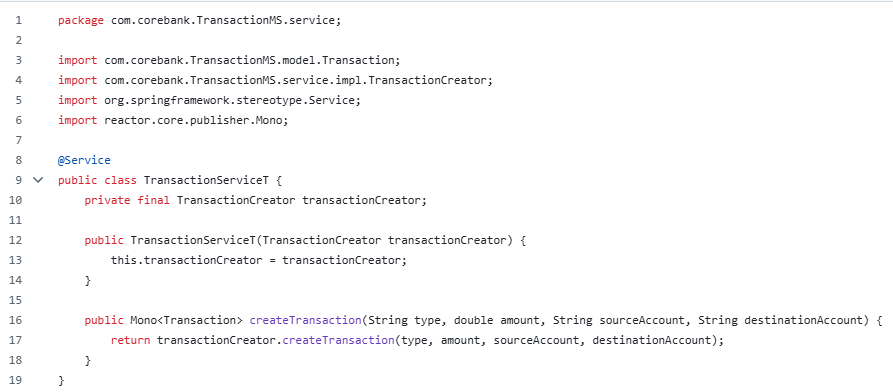
**WithdrawalOperatioImpl**



**AccountServiceT**



**TransactionServiceT**



# Resumen

la refactorización aplicada en TransactionMS ha seguido los principios SOLID para mejorar la modularidad, flexibilidad y mantenibilidad del código. La división de responsabilidades en clases especializadas, la abstracción de operaciones y la segregación de interfaces permiten una arquitectura más escalable y fácil de modificar. Además, la implementación de WebClient optimiza la comunicación con el sistema de cuentas, fortaleciendo la funcionalidad del sistema de transacciones.