Proyecto N° 4

MAPEO DE PRINCIPIOS SOLID Y PATRONES DE DISEÑO EN LA SOLUCIÓN

Elaborado por:

Lisbeth Callata Churata

ÍNDICE

I. INTRODUCCIÓN

Este documento tiene como objetivo mapear los principios SOLID y los patrones de diseño utilizados en el desarrollo de la solución de microservicios para la gestión de cuentas bancarias. A continuación, se describe cómo estos principios fueron aplicados en el diseño y desarrollo del sistema.

II. PRINCIPIOS SOLID

Los principios SOLID son un conjunto de cinco principios de diseño de software que ayudan a crear sistemas más robustos, fáciles de mantener y de extender. A continuación, se describe cómo se aplicaron estos principios en la solución:

2.1 Principio de Responsabilidad Única (Single Responsibility Principle)

Un módulo o clase debe tener una única razón para cambiar, es decir, debe tener una única responsabilidad.

Aplicación en el proyecto:

- **Clases de dominio** como **Account**, **Transaction**, etc., tienen una sola responsabilidad: representar los datos de las cuentas y transacciones. Estas clases no contienen lógica de negocio compleja, sino que se enfocan en su rol como entidades del sistema.
- **Servicios** como **AccountServiceImpl** están encargados exclusivamente de la lógica relacionada con las cuentas, como crear cuentas o consultar el saldo, manteniendo así una única responsabilidad.

Ejemplo:

```java
package com.banco.transacciones.model;

import ...

@Document(collection = "accounts")  48 usages  Lisbeth84 *
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Account {

    @Id
    private String id;
    private String numeroCuenta;
    private Double saldo = 0.0;
    private AccountType tipoCuenta;
    private String clienteId;
}
```

2.2 Principio de Abierto/Cerrado (Open/Closed Principle)

Las clases deben estar abiertas para su extensión, pero cerradas para su modificación.

Aplicación en el proyecto:

- **Interfaz AccountService:** El servicio de cuentas está diseñado de manera que puedes extender sus funcionalidades (por ejemplo, agregar nuevos métodos de negocio) sin modificar la clase base.
- Si se desea agregar nuevos tipos de cuentas, solo se necesita crear una nueva implementación de la interfaz **AccountService** sin modificar el código existente.

Ejemplo:

```java
package com.banco.transacciones.service;

import com.banco.transacciones.model.Account;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface AccountService {  6 usages  1 implementation  Lisbeth84 *

    Mono<Account> getAccount(String id);  6 usages  1 implementation  Lisbeth84
    Mono<Account> createAccount(Account account);  6 usages  1 implementation  Lisbeth84
    Mono<Account> updateAccount(Account account);  3 usages  1 implementation  Lisbeth84
    Flux<Account> getAllAccounts();  1 usage  1 implementation  Lisbeth84
}
```

2.3 Principio de Sustitución de Liskov (Liskov Substitution Principle)

Los objetos de una clase derivada deben poder reemplazar a los objetos de la clase base sin afectar el comportamiento del sistema.

Aplicación en el proyecto:

- En la solución, la clase **AccountServiceImpl** implementa la interfaz **AccountService**. Si en el futuro se crearan nuevas implementaciones de **AccountService** (por ejemplo, para un tipo diferente de almacenamiento de datos), estas implementaciones seguirían cumpliendo con las expectativas de la interfaz y serían intercambiables sin afectar al cliente que utiliza el servicio.

Ejemplo:

```java
package com.banco.transacciones.service.impl;

import com.banco.transacciones.model.Account;
import com.banco.transacciones.repository.AccountRepository;
import com.banco.transacciones.service.AccountService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@{...}
public class AccountServiceImpl implements AccountService {

    private final AccountRepository accountRepository;

    @Override  6 usages  Lisbeth84
    public Mono<Account> getAccount(String id) { return accountRepository.findById(id); }

    @Override  6 usages  Lisbeth84
    public Mono<Account> createAccount(Account account) { return accountRepository.save(account); }

    @Override  3 usages  Lisbeth84
    public Mono<Account> updateAccount(Account account) { return accountRepository.save(account); }

    @Override  1 usage  Lisbeth84
    public Flux<Account> getAllAccounts() { return null; }
}
```

2.4 Principio de Segregación de Interfaces (Interface Segregation Principle)

Los clientes no deben verse obligados a depender de interfaces que no utilizan.

Aplicación en el proyecto:

- Las interfaces están diseñadas de manera granular. Por ejemplo, la interfaz **AccountService** tiene métodos que son relevantes para las operaciones de cuentas, y no se incluyen métodos adicionales que no estén relacionados con la lógica de las cuentas.

Ejemplo:

```java
package com.banco.transacciones.service;

import com.banco.transacciones.model.Account;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface AccountService {  6 usages  1 implementation  Lisbeth84 *

    Mono<Account> getAccount(String id);  6 usages  1 implementation  Lisbeth84
    Mono<Account> createAccount(Account account);  6 usages  1 implementation  Lisbeth84
    Mono<Account> updateAccount(Account account);  3 usages  1 implementation  Lisbeth84
    Flux<Account> getAllAccounts();  1 usage  1 implementation  Lisbeth84
}
```

2.5 Principio de Inversión de Dependencias (Dependency Inversion Principle)

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

Aplicación en el proyecto:

- La clase **AccountServiceImpl** depende de la abstracción **AccountRepository**, que es una interfaz. Esto permite cambiar la implementación del repositorio (por ejemplo, usando una base de datos diferente) sin modificar el código de servicio.
- El servicio no conoce detalles específicos del repositorio, solo interactúa con la interfaz, lo que facilita la extensibilidad y la inyección de dependencias.

Ejemplo:

```
1   package com.banco.transacciones.repository;
2
3   import ...
7
8   @Repository  8 usages  Lisbeth84
9   public interface AccountRepository extends ReactiveMongoRepository<Account, String> {
10      Mono<Account> findByNumeroCuenta(String numeroCuenta);  6 usages  Lisbeth84
11
12  }
```

III. PATRONES DE DISEÑO

En la solución también se han utilizado algunos patrones de diseño que ayudan a mejorar la flexibilidad y la mantenibilidad del código.

3.1 Patrón de Diseño - Singleton

Asegura que una clase tenga una única instancia y proporciona un punto de acceso global a ella.

Aplicación en el proyecto:

El patrón Singleton puede aplicarse en clases que gestionan la configuración global o la conexión a la base de datos, aunque en el contexto del proyecto no se implementa explícitamente un Singleton para los servicios, ya que utilizamos un modelo basado en inyección de dependencias. Sin embargo, el principio detrás de asegurar una única instancia del servicio es utilizado, particularmente en la configuración de Spring.

```java
package com.banco.transacciones.service.impl;

import com.banco.transacciones.model.Account;
import com.banco.transacciones.repository.AccountRepository;
import com.banco.transacciones.service.AccountService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RequiredArgsConstructor  1 usage  Lisbeth84
@Service
public class AccountServiceImpl implements AccountService {

    private final AccountRepository accountRepository;

    @Override  6 usages  Lisbeth84
    public Mono<Account> getAccount(String id) { return accountRepository.findById(id); }

    @Override  6 usages  Lisbeth84
    public Mono<Account> createAccount(Account account) { return accountRepository.save(account); }

    @Override  3 usages  Lisbeth84
    public Mono<Account> updateAccount(Account account) { return accountRepository.save(account); }

    @Override  1 usage  Lisbeth84
    public Flux<Account> getAllAccounts() { return null; }
```

3.2 Patrón de Diseño - Factory

Proporciona una interfaz para crear objetos, pero permite que las clases hijas decidan qué clase instanciar.

Aplicación en el proyecto:

El método **createTransaction** en el servicio **TransactionServiceImpl** es una forma de Factory, ya que crea diferentes tipos de transacciones (depósito, retiro, transferencia) en función del tipo de transacción.

```java
    @Override  11 usages  Lisbeth84
    public Mono<Transaction> createTransaction(Transaction transaction) {
        switch (transaction.getTipo()) {
            case DEPOSITO:
                return createDeposit(transaction);  // Llamamos al método que maneja los depósitos
            case RETIRO:
                return createWithdrawal(transaction);  // Llamamos al método que maneja los retiros
            case TRANSFERENCIA:
                return createTransfer(transaction);  // Llamamos al método que maneja las transferencias
            default:
                return Mono.error(new RuntimeException("Tipo de transacción desconocido"));
        }
    }
```

3.3 Patrón de Diseño - Strategy

Permite definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. El patrón Strategy permite a un cliente elegir el algoritmo que necesita en tiempo de ejecución.

Aplicación en el proyecto:

Si en el futuro se necesitaran diferentes algoritmos para calcular el saldo o aplicar descuentos en diferentes tipos de cuentas, el patrón Strategy podría ser útil para encapsular cada algoritmo y permitir su uso según el tipo de cuenta.

IV. CONCLUSIÓN

En esta solución, hemos aplicado varios principios SOLID para asegurar que el código sea flexible, fácil de extender y de mantener. Además, hemos considerado patrones de diseño como Factory y Strategy, que facilitan la creación y el manejo de objetos en el sistema, y permiten modificar comportamientos sin cambiar el código cliente.

Con estos principios y patrones, la solución es robusta y capaz de adaptarse a futuros cambios y requerimientos sin comprometer la calidad del código.