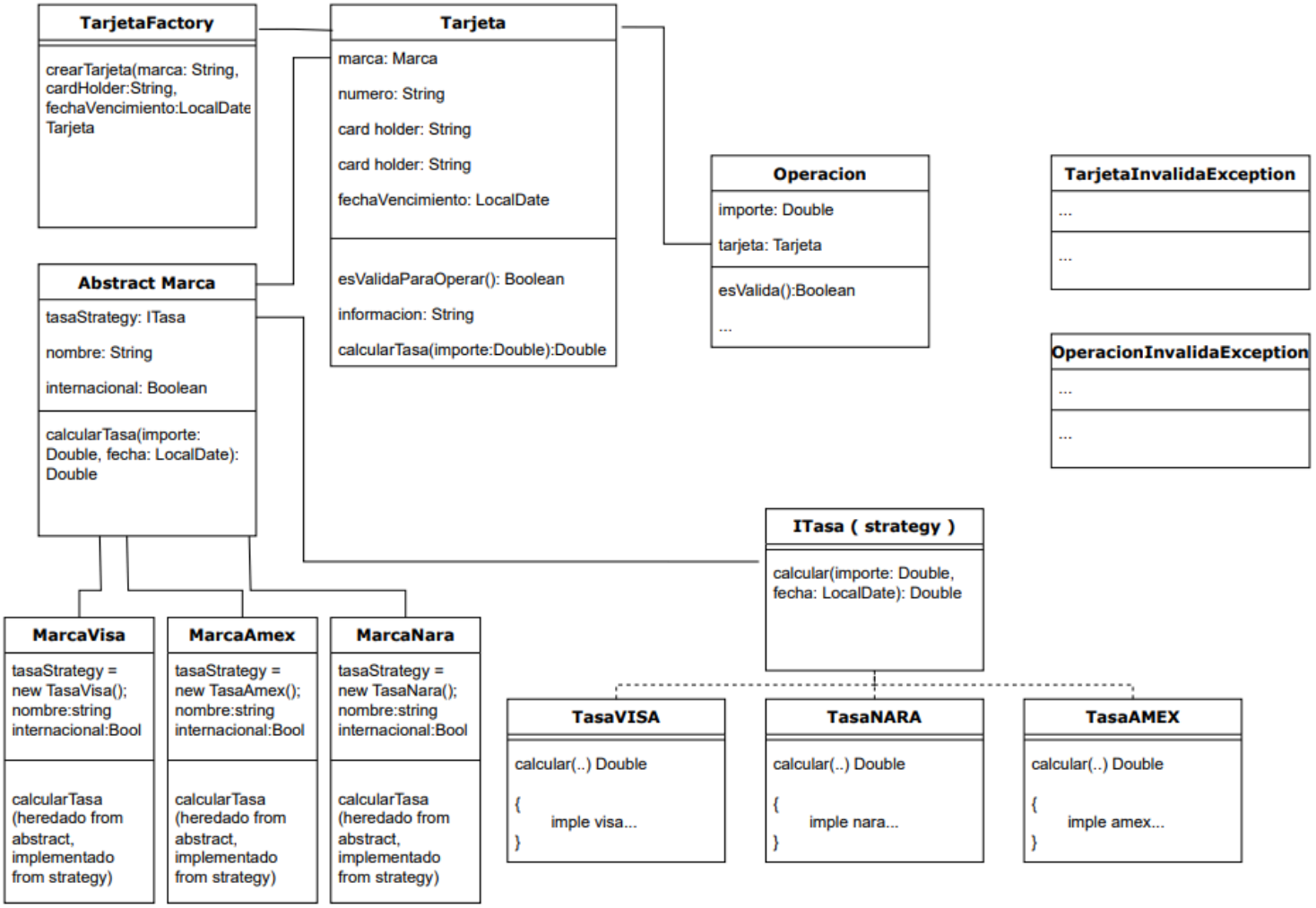
IMPLEMENTACION CHALLENGE TARJETA CREDITOS

by Carlos Esteban Gil

Diagrama de clases (preliminar , de fase de analisis)



Implementacion Concreta en SPringBoot3

(mas completo y ya orientado a su implementacion, con springboot framework)

Modelado del sistema: He diseñado un sistema para gestionar tarjetas de crédito, centrándome en la creación de tarjetas con marcas específicas como Visa, Amex y Nara.

Resumen de la Arquitectura

PreRequisitos para ejecutar la Aplicación:

Mysql 8+ instalado (con user:root pass:root y schema creado creditcardchallenge) + Java 17+ Maven.

(Obs: No hice un docker por cuestion de entregarlo hoy, no lo subí a la nube tambien por cuestiones de tiempo)

0. Configuracion de la aplicacion:

Application.properties:

spring.application.name=demo

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/CreditCardChallenge
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

Configuración de JPA
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

Maven → pom.xml → (ver pom.xml en proyecto)

1. Capas de la Aplicación

Mi aplicación sigue una arquitectura de múltiples capas, que incluye:

- **Entidad (Entity):** Representa las tablas en la base de datos.
- **DTO (Data Transfer Object):** Utilizado para transferir datos entre capas.
- **Servicio (Service):** Contiene la lógica de negocio.
- **Controlador (Controller):** Expone la API REST.
- **Estrategia (Strategy):** Implementa los diferentes algoritmos para calcular la tasa de las distintas marcas de tarjetas.
- **Configuración (Configuration):** Configura los beans y otras configuraciones de la aplicación.

2. Componentes Principales

Main Application:

- `CreditCardChallengeApplication :` Main App. (punto de entrada)

Runners:

- `Pruebas`: → implements CommandLineRunner → se ejecutan los casos de prueba del challenge en forma automática al startear el sistema.
- `DatabaseInitializer`: Se ejecuta antes que pruebas. Se encarga de asegurarse que existan las tarjetas que requiere el sistema para funcionar (visa,nara,amex) con sus respectivos id y datos. (todo automático. en configuracion del sistema, pre-requisito logico para funcionar)

Entidades y DTOs:

- `CreditCardEntity`: Representa una tarjeta de crédito en la base de datos.
- `BrandEntity`: Representa una marca de tarjeta de crédito.
- `CreditCardDTO`: Transferencia de datos de tarjeta de crédito.
- `VisaCreditCardDTO`, `AmexCreditCardDTO`, `NaraCreditCardDTO`: Subclases de `CreditCardDTO` para cada tipo de tarjeta.

Servicios:

- `TarjetaService`: Contiene la lógica para gestionar tarjetas de crédito.
- `BrandService`: solo define la interface para sus implementaciones visaservice,naraservice etc
- `VisaService`, `AmexService`, `NaraService`: Servicios específicos para cada marca de tarjeta. Obs: son implementaciones de BarandService no de tarjeta service.
- `OperacionService`: Gestiona las operaciones y sus tasas (que dependen de la tarjeta asociada que dependen del brand de la tarjeta que dependen de la implementacion del strategy concreta que se le tiene asociada)

Estrategias:

- `ITasaStrategy`: Interfaz que define el método `calcular`.
- `TasaVisaStrategy`, `TasaAmexStrategy`, `TasaNaraStrategy`: Implementaciones concretas de `ITasaStrategy` para cada marca de tarjeta.

Inicio del sistema:

El sistema se inicia cuando se ejecuta la aplicación principal, CreditCardChallengeApplication.

(CreditCardChallengeApplication.java → run as: java application)

Ejecución de la prueba: (Pruebas.java se ejecuta auto ya que implementa commandlinerunner)

(*Obs: Pruebas.java contiene todos los casos de prueba solicitados en el doc del challenge)*

Por ejemplo:

Dentro del método ejecutarPruebas(), creé una instancia del factory de tarjetas de crédito y usé este factory para crear una tarjeta de crédito específica, como Visa.

El factory determina qué servicio utilizará para obtener la marca correspondiente y crea el DTO de la tarjeta de crédito con la marca asociada.(y ademas, dá de alta en db la tarjeta si no existia)

Luego Creé un objeto de modelo de negocio desde ese dto que me brinda el factory mediante una utilidad sencilla de mapeo de creditCardDTO a una instancia de Tarjeta (del modelo)

Luego imprimí todos los datos de la tarjeta ya creada y cargada en db por el factory

Luego implementé todos los casos de prueba solicitados en el doc del challenge (y comenté cada uno para que se entienda bien a simple vista la relacion con lo solicitado)

Conceptos utilizados:

Patrón de diseño Factory: Utilizado para crear instancias de DTOs de tarjetas de crédito de manera genérica.

Inyección de dependencias (DI): Utilizada para proporcionar instancias de servicios a los componentes que las necesitan.

DTOs: Utilizados para transferir datos entre capas de la aplicación.

Interfaces: Utilizadas para definir contratos comunes entre los servicios de marca de tarjeta.

Genericidad: Utilizada en el factory para permitir la creación de instancias de DTOs de diferentes tipos.

Buena práctica de diseño: Dividí la funcionalidad en componentes independientes y cohesivos para facilitar el mantenimiento y la escalabilidad.

Ventajas:

Flexibilidad: El sistema es flexible y extensible, ya que permite agregar nuevos tipos de tarjetas de crédito y servicios de marca fácilmente.

Separación de preocupaciones: La lógica relacionada con la creación de tarjetas de crédito y la obtención de marcas se encuentra en componentes separados, lo que facilita la comprensión y el mantenimiento del código.

Desventajas:

1. Separation of Concerns: Cada clase tiene una responsabilidad clara.
2. Spring Boot & DI: Uso de Spring Boot y Dependency Injection facilita la gestión de dependencias y el ciclo de vida de los beans.

3. Modularidad: Fácil de añadir nuevas marcas o modificar la lógica existente sin impactar otras partes del sistema.
4. Testing: La estructura facilita el testeo unitario y de integración.

Contras:

1. Complejidad Inicial: Puede ser complicado para desarrolladores nuevos en Spring y en arquitectura orientada a servicios.
2. Overhead de Configuración: Configuración adicional para repositorios, servicios y fábricas.
3. Manejo de Excepciones: Necesidad de un manejo robusto de excepciones para evitar fallos en tiempo de ejecución.

Resumen Final

En resumen, he diseñado un sistema flexible y modular para gestionar tarjetas de crédito, utilizando buenas prácticas de diseño y patrones de programación para garantizar la escalabilidad y la mantenibilidad del código.

Esta implementación proporciona una estructura clara y extensible para gestionar tarjetas de crédito y sus marcas utilizando Spring Boot. La separación en servicios y fábricas permite una fácil escalabilidad y mantenibilidad. El uso de repositorios JPA garantiza una integración fluida con la base de datos. Aunque hay un overhead inicial en la configuración y comprensión del modelo, los beneficios a largo plazo en términos de modularidad y capacidad de prueba son significativos.