UNIVERSIDADE FEDERAL DO CEARÁ

Arquitetura e Padrões de Projeto – NexuBank

Projeto Detalhado de Software - 02A - 2024.2

Prof.ª Paulyne Matthews Jucá

Autor

Daniel Santos Fernandes

Sumário

Domínio

O sistema aborda o domínio financeiro e bancário, focando na gestão de contas, usuários e transações. Nele, os principais elementos de domínio incluem:

- **Contas Bancárias:** São gerenciadas diversas modalidades, como contas correntes, poupança e empresariais, cada uma com suas regras específicas para limites, rendimentos e operações.
- **Usuários:** O sistema diferencia pessoas físicas e jurídicas, tratando as particularidades de cada um (CPF para pessoas físicas e CNPJ para pessoas jurídicas).
- **Operações Financeiras:** São implementadas transações diversas, como depósitos, transferências via PIX, operações com cartão (compra e pagamento de fatura) e cálculos de rendimento para poupança.
- **Chaves PIX:** Há um módulo dedicado ao gerenciamento de chaves PIX, permitindo a criação, validação e remoção de chaves (por exemplo, CPF, CNPJ, email, telefone ou chave aleatória) para facilitar as operações digitais.

Arquitetura do Projeto

Decisão de Aplicação:
A arquitetura do NexuBank foi projetada para promover modularidade, escalabilidade e facilidade de manutenção. O sistema é dividido em camadas bem definidas, permitindo que cada parte seja desenvolvida, testada e evoluída de forma independente. Essa separação de responsabilidades garante que a interface com o usuário seja simples, enquanto a complexidade das operações de negócio, acesso a dados e integrações é gerenciada internamente pelas camadas especializadas.

Implementação:

- **Camada de Apresentação:**
  Os controladores REST expõem os serviços do sistema, recebendo as requisições dos clientes, realizando validações iniciais (por meio de DTOs) e encaminhando os dados para a camada de serviço. Essa camada é responsável por transformar as solicitações externas em chamadas aos processos internos, mantendo a comunicação clara e padronizada.
- **Camada de Serviço:**
  Essa camada centraliza a lógica de negócio, orquestrando as operações relacionadas a contas, transações e usuários. Ela integra diversas funcionalidades, como validação de dados, conversão entre modelos de domínio e DTOs, e coordenação das operações de persistência e notificações.
- **Camada de Domínio:**
  Nesta camada, as entidades representam os conceitos fundamentais do negócio, como contas, transações, usuários e chaves Pix. As regras de negócio e os comportamentos essenciais são implementados aqui, garantindo que todas as operações reflitam as necessidades reais do sistema financeiro.
- **Camada de Persistência:**
  O acesso aos dados é gerenciado por repositórios que abstraem as operações de criação, leitura, atualização e exclusão. Essa camada lida com a comunicação com o banco de dados, permitindo que a lógica de negócio permaneça desacoplada dos detalhes de armazenamento, o que facilita a manutenção e a evolução do sistema.

Padrões de Projeto

Factory Method Pattern

Decisão de Aplicação:

O Factory Method Pattern foi utilizado para centralizar e encapsular a criação de objetos complexos, evitando que a lógica de instanciamento se espalhe pelo sistema. Dessa forma, a criação de contas e de usuários é feita de forma flexível e desacoplada, permitindo a adição de novos tipos sem a necessidade de alterar o código cliente.

Implementação:

- **Contas:**
  - A interface ContaCreator define o contrato para a criação de contas.
  - As classes concretas, como ContaCorrenteCreator, ContaEmpresarialCreator e ContaPoupancaCreator, implementam esse contrato, cada uma encapsulando a lógica de criação específica para cada tipo de conta.
  - A configuração das fábricas é centralizada em ContaFactoryConfig, que mapeia as implementações por meio do enum TipoConta.
  - A resolução é feita através da classe ContaFactoryResolver, que seleciona a fábrica adequada conforme o tipo de conta solicitado.
- **Usuários:**
  - A criação de usuários segue uma abordagem similar com a interface UsuarioCreator.
  - As classes UsuarioPessoaFisicaCreator e UsuarioPessoaJuridicaCreator realizam a criação de instâncias de PessoaFisica ou PessoaJuridica, conforme estratégia definida (CPF ou CNPJ).
  - A classe UsuarioFactory atua como coordenadora, delegando a criação para a implementação apropriada com base no documento fornecido.
- **Chaves Pix:**
  - Uma interface, ChavePixCreator, define o contrato para a criação da chave PIX.
  - Diversas implementações concretas (como ChavePixCpfCreator, ChavePixCnpjCreator, ChavePixEmailCreator, ChavePixTelefoneCreator e ChavePixAleatoriaCreator) encapsulam a lógica específica de criação para cada tipo de chave.

- A classe ChavePixFactory atua como resolvedor, selecionando e retornando a instância adequada de ChavePixCreator com base no tipo de chave (definido pelo enum ChavePixTipo)

Observer Pattern

Decisão de Aplicação:

O Observer Pattern foi adotado para permitir que diferentes partes do sistema sejam notificadas sobre eventos relevantes, como operações que alteram o estado de uma conta. Essa abordagem promove o desacoplamento entre o componente que gera o evento e os componentes que reagem a ele.

Implementação:

- A interface ContaObserver define o contrato que os observadores devem seguir para reagir aos eventos.
- A classe ContaEvent encapsula informações sobre eventos ocorridos em uma conta.
- A classe ContaNotifier gerencia a lista de observadores e os notifica sempre que um evento ocorre.
- O enum EventType define os tipos de eventos observados, como TRANSFERENCIA_PIX e RECEBIMENTO_PIX.
- A classe NotificacaoService implementa o comportamento específico de notificação quando um evento é disparado.

Chain of Responsibility Pattern

Decisão de Aplicação:

O Chain of Responsibility Pattern foi utilizado para gerenciar a validação de transferências via PIX, permitindo que múltiplas regras de negócio sejam avaliadas de forma sequencial e modular.

- A interface PixTransferValidationHandler define o contrato para os validadores da cadeia.
- SaldoValidator verifica se a conta possui saldo suficiente para a transferência.
- ValorPositivoValidator assegura que o valor da transferência seja positivo.

- ChaveDestinoValidator valida se a chave Pix de destino existe e está vinculada a uma conta.
- A classe PixTransferValidationChain gerencia e executa a cadeia de validadores em sequência, garantindo que todas as regras sejam aplicadas antes da conclusão da transferência.

Strategy Pattern

Decisão de Aplicação:

O Strategy Pattern foi implementado para desacoplar a lógica de diferentes tipos de transações, garantindo que cada uma seja processada de acordo com suas regras específicas. Além disso, ele é utilizado para definir a estratégia para criação dos diferentes tipos de usuários. Esse padrão permite flexibilidade na escolha da estratégia de execução, facilitando a adição de novos tipos de transação ou usuários sem impactar o código existente.

- **Transações:**
    - A interface TransacaoStrategy define o contrato para todas as estratégias de transação.
    - PixTransacaoStrategy implementa a lógica para transferências e recebimentos via Pix.
    - CartaoTransacaoStrategy gerencia a lógica de compras e pagamentos via cartão.
    - A classe TransacaoContext é responsável por selecionar e executar a estratégia apropriada com base no tipo de transação.
- **Usuários:**
    - A interface UsuarioStrategy define o contrato para a criação de pessoa física e pessoa jurídica.
    - UsuarioPessoaFisicaStrategy implementa a estratégia para a criação de usuários com do tipo pessoa física.
    - Enquanto UsuarioPessoaJuridicaStrategy implementa a estratégia para a criação de usuários com do tipo pessoa jurídica.

Data Transfer Object (DTO) Pattern

Decisão de Aplicação:
Para separar a lógica de apresentação e transporte dos dados da lógica de domínio, o padrão DTO foi empregado. Essa abordagem facilita a comunicação entre as camadas de aplicação e persistência, permitindo que os dados sejam transferidos de forma estruturada e segura.

Implementação:

- Foram definidas classes específicas para a criação, atualização e resposta (por exemplo, para contas, transações e usuários), que encapsulam os dados necessários em cada operação.
- Isso garante que a camada de apresentação não dependa diretamente das entidades de domínio, promovendo maior desacoplamento e facilitando testes e manutenções.

Data Mapper Pattern

Decisão de Aplicação:
O padrão Data Mapper foi utilizado para isolar a lógica de conversão entre os objetos de domínio e os DTOs. Dessa forma, a transformação dos dados para formatos adequados a cada camada é centralizada e reutilizável.

Implementação:

- Foram implementadas classes mapeadoras responsáveis por converter entidades para DTOs e vice-versa.
- Essa separação permite que mudanças no modelo de dados ou na forma de apresentação sejam gerenciadas sem afetar a lógica de negócio.

Repository Pattern

Decisão de Aplicação:
Para abstrair o acesso a dados e encapsular as operações de persistência, o padrão Repository foi adotado. Essa abordagem promove um acesso desacoplado aos dados,

facilitando a manutenção e a evolução do sistema sem que a lógica de negócio precise conhecer detalhes da tecnologia de armazenamento.

Implementação:

- Interfaces específicas para cada tipo de entidade (como contas, usuários e chaves Pix) definem os métodos de CRUD e consultas customizadas.
- A utilização do framework de persistência (Spring Data JPA, por exemplo) possibilita que a implementação concreta das operações seja gerenciada de forma transparente, mantendo o código de negócio limpo e focado nas regras de negócio.