Trabalho de Orientação a Objetos.

Documentação do projeto Bank

Professor:
Gleiph

Juiz de Fora

Março de 2025

1 Visão Geral do Sistema

O sistema bancário é uma aplicação Java que implementa funcionalidades básicas de um banco, incluindo o gerenciamento de usuários (clientes, caixas e gerentes), contas bancárias e investimentos. O sistema utiliza conceitos de orientação a objetos como herança, polimorfismo e encapsulamento para modelar as entidades do domínio bancário.

2 Estrutura do Projeto

O projeto está organizado nos seguintes pacotes:

- `com.mycompany.bank`: Pacote principal que contém a classe `Main`.
- `com.mycompany.bank.model`: Contém as classes de modelo do sistema.
- `com.mycompany.bank.service`: Contém a lógica de negócio do sistema.
- `com.mycompany.bank.dao`: Contém classes para persistência de dados.
- `com.mycompany.bank.exceptions`: Contém as exceções personalizadas.
- `com.mycompany.bank.gui`: Contém as classes de interface gráfica.

3 Classes e Interfaces

3.1 Pacote `model`

3.1.1 Interface `Autenticavel`

Descrição: Define um contrato para autenticação de usuários.

Métodos:

- `boolean autenticar(String senha)`: Verifica se a senha fornecida é válida.

3.1.2 Classe `Usuario` (Abstrata)

Descrição: Classe base para todos os tipos de usuários do sistema.

Atributos:

- `id`: Identificador único do usuário (gerado automaticamente).
- `nome`: Nome do usuário.
- `cpf`: CPF do usuário.
- `senha`: Senha do usuário.

Métodos:

- `boolean autenticar(String senha)`: Implementação do método da interface `Autenticavel`.
- Getters e setters para todos os atributos.

3.1.3 Classe `Cliente`

Descrição: Representa um cliente do banco.

Herança: Estende a classe `Usuario`.

Atributos:

- `contas`: Lista de contas bancárias do cliente.

Métodos:

- `void adicionarConta(Conta conta)`: Adiciona uma conta à lista de contas do cliente.
- `double consultarSaldo(String numeroConta)`: Consulta o saldo de uma conta específica.
- `void transferir(String contaOrigem, String contaDestino, double valor)`: Realiza transferência entre contas.

3.1.4 Classe `Caixa`

Descrição: Representa um funcionário caixa do banco.

Herança: Estende a classe `Usuario`.

Métodos:

- `void processarDeposito(Conta conta, double valor)`: Processa um depósito em uma conta.
- `void processarSaque(Conta conta, double valor)`: Processa um saque de uma conta.
- `void processarTransferencia(Conta origem, Conta destino, double valor)`: Processa uma transferência entre contas.

3.1.5 Classe Gerente

Descrição: Representa um gerente do banco.

Herança: Estende a classe `Usuario`.

Atributos:

- `opcoesInvestimento`: Lista de investimentos disponíveis.

Métodos:

- `void adicionarInvestimento(Investimento inv)`: Adiciona um investimento à lista.
- `void removerInvestimento(String nome)`: Remove um investimento da lista.
- `boolean avaliarCredito(Cliente cliente, double valorSolicitado)`: Avalia se um cliente pode receber crédito.

3.1.6 Classe Conta

Descrição: Representa uma conta bancária.

Atributos:

- `numero`: Número da conta.
- `saldo`: Saldo atual da conta.

Métodos:

- `void depositar(double valor)`: Adiciona valor ao saldo da conta.
- `void sacar(double valor)`: Retira valor do saldo da conta.
- `void transferir(Conta destino, double valor)`: Transfere valor para outra conta.

3.1.7 Classe Investimento (Abstrata)

Descrição: Classe base para todos os tipos de investimentos.

Atributos:

- `nome`: Nome do investimento.
- `descricao`: Descrição do investimento.

- `valorMinimo`: Valor mínimo para aplicação.

Métodos:

- `void aplicar(double valor)`: Método abstrato para aplicar em um investimento.
- `void resgatar(double valor)`: Método abstrato para resgatar de um investimento.

3.1.8 Classe RendaFixa

Descrição: Representa um investimento de renda fixa.

Herança: Estende a classe `Investimento`.

Atributos:

- `taxaRendimento`: Taxa de rendimento do investimento.
- `prazoMinimo`: Prazo mínimo em dias.
- `prazoMaximo`: Prazo máximo em dias.

Métodos:

- `void aplicar(double valor)`: Implementação para aplicar em renda fixa.
- `void resgatar(double valor)`: Implementação para resgatar de renda fixa.

3.1.9 Classe RendaVariavel

Descrição: Representa um investimento de renda variável.

Herança: Estende a classe `Investimento`.

Atributos:

- `percentualRisco`: Percentual de risco do investimento.
- `rentabilidadeEsperada`: Rentabilidade esperada do investimento.

Métodos:

- `void aplicar(double valor)`: Implementação para aplicar em renda variável.
- `void resgatar(double valor)`: Implementação para resgatar de renda variável.

3.2 PACOTE exceptions

3.2.1 CLASSE SaldoInsuficienteException

Descrição: Exceção lançada quando não há saldo suficiente para uma operação.

Herança: Estende `Exception`.

3.2.2 CLASSE SenhaInvalidaException

Descrição: Exceção lançada quando a senha informada não é válida.

Herança: Estende `Exception`.

3.2.3 CLASSE UsuarioNaoEncontradoException

Descrição: Exceção lançada quando um usuário não é encontrado.

Herança: Estende `Exception`.

3.3 PACOTE service

3.3.1 CLASSE SistemaBancario

Descrição: Classe de serviço que gerencia as listas de usuários, contas e investimentos.

Atributos:

- `usuarios`: Lista de usuários do sistema.
- `contas`: Lista de contas do sistema.
- `investimentos`: Lista de investimentos disponíveis.

Métodos:

- `void adicionarUsuario(Usuario u)`: Adiciona um usuário ao sistema.
- `Usuario buscarUsuarioPorCpf(String cpf)`: Busca um usuário pelo CPF.
- `void removerUsuario(String cpf)`: Remove um usuário do sistema.
- `Usuario login(String cpf, String senha)`: Realiza login de um usuário.
- `void adicionarConta(Conta c)`: Adiciona uma conta ao sistema.

- `Conta buscarContaPorNumero(String numero)`: Busca uma conta pelo número.
- `void adicionarInvestimento(Investimento i)`: Adiciona um investimento ao sistema.

3.4 Pacote `dao`

3.4.1 Classe `UsuarioDAO`

Descrição: Classe para persistência de dados de usuários.

Métodos:

- `void salvarEmJson(String caminho, List<Usuario> usuarios)`: Salva lista de usuários em JSON.
- `List<Usuario> carregarDeJson(String caminho)`: Carrega lista de usuários de JSON.
- `void salvarEmXml(String caminho, List<Usuario> usuarios)`: Salva lista de usuários em XML.
- `List<Usuario> carregarDeXml(String caminho)`: Carrega lista de usuários de XML.

3.4.2 Classe `UsuariosWrapper`

Descrição: Classe wrapper para serializar/deserializar lista de usuários em XML via JAXB.

Atributos:

- `usuarios`: Lista de usuários.

Métodos:

- Getters e setters para o atributo `usuarios`.

3.5 Pacote `gui`

3.5.1 Classe `MainView`

Descrição: Janela principal do sistema para login.

Herança: Estende `JFrame`.

Atributos:

- `cpfField`: Campo para entrada do CPF.
- `senhaField`: Campo para entrada da senha.

- `loginButton`: Botão para realizar login.
- `sistema`: Referência ao sistema bancário.

Métodos:

- Construtor que inicializa a interface gráfica.
- `ActionListener` para o botão de login.

3.6 Pacote Principal

3.6.1 Classe Main

Descrição: Classe principal que inicia o sistema.

Métodos:

- `main(String[] args)`: Método principal que cria o sistema, adiciona usuários de exemplo e inicia a GUI.

4 Fluxo de Funcionamento

O sistema é iniciado pela classe `Main`. São criados usuários de exemplo (cliente, caixa e gerente) e uma conta é criada para o cliente. Em seguida, a interface gráfica é iniciada, permitindo que o usuário faça login com CPF e senha. Dependendo do tipo de usuário, diferentes funcionalidades estarão disponíveis:

- **Cliente:** Consultar saldo e transferir dinheiro.
- **Caixa:** Processar depósitos, saques e transferências.
- **Gerente:** Gerenciar investimentos e avaliar crédito.

5 Padrões de Projeto Utilizados

- **Herança:** Utilizada para especializar os tipos de usuários (`Cliente`, `Caixa`, `Gerente`) a partir da classe base `Usuario`.
- **Polimorfismo:** Utilizado na interface `Autenticavel` e nas classes de investimento.
- **DAO (Data Access Object):** Utilizado para separar a lógica de acesso a dados do resto da aplicação.

6 TESTES

Esta seção irá documentar os testes unitários e de integração implementados para o sistema bancário.

6.1 ESTRUTURA DE TESTES

O projeto utiliza JUnit como framework de testes e está organizado nas seguintes classes de teste:

- `OperacoesFinanceirasTest` - Testes de operações financeiras básicas
- `InvestimentoTest` - Testes para os tipos de investimentos
- `SolicitacaoCreditoTest` - Testes para solicitações de crédito
- `SolicitacaoCreditoIntegracaoTest` - Testes de integração para fluxos de solicitação de crédito
- `IntegracaoSistemaBancarioTest` - Testes de integração do sistema bancário completo
- `TestDataFactory` - Classe utilitária para criação de dados de teste

6.2 TESTES UNITÁRIOS

6.2.1 INVESTIMENTOTEST

Testes para as classes de investimento (Renda Fixa e Variável).

Teste	**Descrição**	**Resultado Esperado**
testAplicacaoRendaFixaComValorValido	Verifica se uma aplicação com valor válido em renda fixa é aceita	Não deve lançar exceção
testAplicacaoRendaFixaComValorAbaixoDoMinimo	Verifica se uma aplicação abaixo do valor mínimo é rejeitada	Deve lançar `IllegalArgumentException`
testPropriedadesRendaFixa	Verifica as propriedades de um investimento de renda fixa	Propriedades devem corresponder aos valores definidos
testAplicacaoRendaVariavelComValorValido	Verifica se uma aplicação com valor válido em renda variável é aceita	Não deve lançar exceção
testAplicacaoRendaVariavelComValorAbaixoDoMinimo	Verifica se uma aplicação abaixo do valor mínimo é rejeitada	Deve lançar `IllegalArgumentException`
testPropriedadesRendaVariavel	Verifica as propriedades de um investimento de renda variável	Propriedades devem corresponder aos valores definidos
testResgateRendaFixa	Verifica o resgate de investimento em renda fixa	Não deve lançar exceção
testResgateRendaVariavel	Verifica o resgate de investimento em renda variável	Não deve lançar exceção

Tabela 1: Testes da classe `InvestimentoTest`

6.2.2 OPERACOESFINANCEIRASTEST

Testes para operações financeiras básicas (saque, depósito e transferência).

Teste	Descrição	Resultado Esperado
testDepositoValido	Verifica se um depósito válido é processado corretamente	Saldo deve aumentar e histórico deve ser atualizado
testDepositoInvalido	Verifica se um depósito com valor negativo é rejeitado	Deve lançar `IllegalArgumentException`
testSaqueValido	Verifica se um saque válido é processado corretamente	Saldo deve diminuir e histórico deve ser atualizado
testSaqueComSaldoInsuficiente	Verifica se um saque com saldo insuficiente é rejeitado	Deve lançar `SaldoInsuficienteException`
testSaqueComValorNegativo	Verifica se um saque com valor negativo é rejeitado	Deve lançar `IllegalArgumentException`
testTransferenciaValida	Verifica se uma transferência válida é processada corretamente	Saldos de ambas as contas devem ser atualizados, assim como o histórico
testTransferenciaComSaldoInsuficiente	Verifica se uma transferência com saldo insuficiente é rejeitada	Deve lançar `SaldoInsuficienteException`
testConsultaSaldo	Verifica se a consulta de saldo retorna o valor correto	Saldo retornado deve ser o valor esperado
testPertenceConta	Verifica se a verificação de propriedade da conta funciona corretamente	Deve retornar verdadeiro para contas do cliente e falso para outras contas

Tabela 2: Testes da classe `OperacoesFinanceirasTest`

6.2.3 SolicitacaoCreditoTest

Testes para solicitações de crédito.

Teste	Descrição	Resultado Esperado
testCriacaoSolicitacaoCredito	Verifica se uma solicitação de crédito é criada corretamente	Propriedades devem corresponder aos valores definidos
testAnaliseAprovacaoCredito	Verifica o processo de análise e aprovação de crédito	Os estados de análise, aprovação e aceitação devem ser atualizados
testIdUnico	Verifica se cada solicitação tem um ID único	IDs devem ser diferentes para cada solicitação
testFluxoCompleto	Verifica um fluxo completo de solicitação, aprovação e aceitação	Estados devem ser atualizados corretamente em cada etapa
testMultiplasSolicitacoesCliente	Verifica múltiplas solicitações para o mesmo cliente	Todas as solicitações devem ter o mesmo ID de cliente, mas IDs diferentes

Tabela 3: Testes da classe `SolicitacaoCreditoTest`

6.3 Testes de Integração

6.3.1 SolicitacaoCreditoIntegracaoTest

Testes de integração para o fluxo completo de solicitações de crédito.

Teste	Descrição	Resultado Esperado
testProcessoCompletoDeSolicitacaoCredito	Verifica o processo completo de solicitação, análise, aprovação e aceitação de crédito	Estados devem ser atualizados corretamente em cada etapa
testAnaliseMultiplasSolicitacoes	Verifica a análise de múltiplas solicitações pelo gerente	Gerente deve conseguir analisar e aprovar/rejeitar múltiplas solicitações
testClienteAceitaApenasAlgumasSolicitacoes	Verifica a aceitação seletiva de solicitações pelo cliente	Cliente deve conseguir aceitar algumas solicitações e rejeitar outras
testVerificacaoPropriedadesSolicitacao	Verifica as propriedades de cada solicitação	Propriedades devem corresponder aos valores definidos e IDs devem ser únicos

Tabela 4: Testes da classe `SolicitacaoCreditoIntegracaoTest`

6.3.2 IntegracaoSistemaBancarioTest

Testes de integração do sistema bancário completo.

Teste	Descrição	Resultado Esperado
testFluxoOperacoesCompleto	Verifica um fluxo completo de operações, incluindo depósito, saque, transferência, investimentos e solicitação de crédito	Todas as operações devem ser realizadas com sucesso e os saldos devem ser atualizados corretamente
testFluxoMultiplasContasCliente	Verifica operações entre múltiplas contas do mesmo cliente	Transferências entre contas do mesmo cliente devem ser realizadas com sucesso
testFluxoInvestimentos	Verifica aplicações em múltiplos investimentos	Aplicações e resgates devem ser realizados com sucesso

Tabela 5: Testes da classe `IntegracaoSistemaBancarioTest`

6.4 Classe Utilitária

6.4.1 4.1. TestDataFactory

Classe utilitária para criação de dados de teste.

Método	Descrição	Dados Criados
criarSistemaBancarioComUsuarios	Cria um sistema bancário com usuários padrão para testes	Sistema com clientes, gerentes, caixas e contas
criarRendasFixas	Cria uma variedade de investimentos de renda fixa para testes	Array de investimentos de renda fixa
criarRendasVariaveis	Cria uma variedade de investimentos de renda variável para testes	Array de investimentos de renda variável
criarSolicitacoesCredito	Cria solicitações de crédito para um cliente específico	Array de solicitações de crédito

Tabela 6: Testes da classe `TestDataFactory`

6.5 Cobertura de Testes

Os testes cobrem as seguintes funcionalidades principais do sistema:

- **Contas e Operações Bancárias:**
  - Criação de contas
  - Depósitos e saques
  - Transferências entre contas
  - Consulta de saldo
  - Verificação de propriedade de conta
- **Investimentos:**
  - Criação de investimentos de renda fixa e variável
  - Aplicação em investimentos
  - Resgate de investimentos
  - Verificação de propriedades dos investimentos
- **Solicitações de Crédito:**
  - Criação de solicitações
  - Análise e aprovação pelo gerente
  - Aceitação pelo cliente
  - Fluxo completo do processo
- **Integração entre Componentes:**
  - Interação entre usuários, contas e operações
  - Fluxos completos de transações bancárias
  - Múltiplas operações em sequência

7 CONCLUSÃO

O sistema bancário implementa uma arquitetura orientada a objetos com uma separação clara de responsabilidades entre as camadas de modelo, serviço, persistência e interface gráfica. A estrutura do código permite fácil manutenção e extensão para adicionar novas funcionalidades.