A concepção de um Sistema de Gerenciamento Bancário baseado nos moldes e boas práticas da Programação Orientada a Objetos

1st Lucas Pimenta Braga
Dep. de Engenharia de Sistemas
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
2023034552

2nd Mateus de Souza Gontijo
Dep. de Engenharia de Sistemas
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
2020053530

3rd Victor Gabriel dos Santos Silva
Dep. de Engenharia de Sistemas
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
2023060804

***Abstract*—O presente relatório detalha a concepção e implementação de um sistema de gerenciamento bancário, desenvolvido em Python e criado como um projeto da disciplina Análise, Projeto e Programação Orientada a Objetos na Universidade Federal de Minas Gerais, ministrada pela professora Gabriela Nunes. O sistema visa simular operações fundamentais de uma instituição bancária, como o cadastro de clientes, criação de contas e realização de transações bancárias. Nesse sentido, conceitos fundamentais da POO foram utilizados, tais como a abstração, o encapsulamento, a herança e o polimofismo. Esses conceitos foram aplicados visando a criação de um sistema modular, reutilizável e com a idéia de "responsabilidade única", produzindo uma solução modular e funcional para a simulação de um ambiente bancário virtual.**

***Index Terms*—Sistema Bancário, Arquitetura em Camadas, Programação Orientada a Objetos, Python.**

I. Introdução

Nas últimas décadas, o setor bancário tem experimentado uma transformação sisgnificativa, impulsionada pela evolução tecnológica e pela digitalização dos serviços financeiros. A demanda crescente por soluções digitais seguras e eficientes têm levado as instituições financeiras a investir massivamente em tecnologia para atender às necessidades e expectativas dos clientes por maior agilidade e facilidade no acesso aos produtos financeiros [1]. No Brasil, essa transformação é notável, visto a adoção de serviços digitais bancários por uma parcela significativa da população. Estudos indicam que mais de 70 por cento dos brasileiros utilizam serviços bancários digitais, refletindo a rápida digitalização do setor no país [2].

Nesse contexto, a capacidade de desenvolver sistemas que gerenciem operações complexas de forma confiável e escalável torna-se um aspecto essencial no uso de novas tecnologias. A Progrmação Orientada a Objetos (POO) detaca-se como um paradigma essencial, oferecendo conceitos como abstração, encapsulamento, herança e polimorfismo. Tais conceitos facilitam a construção de software modular, reutilizável e de fácil manutenção [3]. Por outro lado, encontra-se na idéia de "Front-end", que está nuito relacionado com a interface gráfica do projeto e é nele onde se desenvolve a aplicação com a qual o usuário irá interagir diretamente, seja em softwares, sites, aplicativos, etc [4]. Portanto, é essencial que o desenvolvedor tenha uma preocupação com a experiência do usuário, uma vez que uma interface gráfica intuitiva, chamativa e elegante influenciam diretamente na satisfação, retenção e fidelização de possíveis clientes. Assim, este relatório visa elucidar os conhecimentos e conceitos aplicados ao longo do desenvolvimento do projeto, assim como o processo de elaboração dos módulos, a integração com a interface gráfica utilizada pelo sistema e os resultados obtidos. O código, assim como o link do seu diagrama UML associado [5], encontra-se disponível em um repositório do GitHub [6].

II. Metodologias

A concretização do sistema de gerenciamento bancário descrito nesse relatório foi norteada por um conjunto de metodologias e princípios de engenharia de software visando a modularidade, flexibilidade e a manutenção da aplicação. O uso de tais práticas foi impulsionado a partir de uma sugestão crítica realizada pela professora Gabriela Nunes, docente responsável pela disciplina, durante uma avaliação preliminar do código, momento em que foi identificado uma interdependência excessiva entre os módulos e classes utilizadas, algo que poderia vir a limitar a eficiência no desenvolvimento de código e prejudicar a fluidez da funcionalidade da aplicação em futuras expansões. Visando contornar essa situação, foi realizada uma reestruturação baseada em Arquitetura em Camadas e Padrões de Projeto Consolidados [7], algo que permitiu a separação de responsabilidades e uma maior fluidez no desenvolvimento e na compreensão da lógica de funcionamento do programa.

A seguir, serão detalhadas as principais abordagens metodológicas e ferramentas adotadas.

A. Arquitetura

A escolha de um modelo de arquitetura para se basear foi o ponto de partida para o desenvolvimento rápido e contínuo do programa, uma vez que permitiu separar o código por atribuições/funções, facilitando o processo lógico de criação.

Foram utilizadas duas abordagens para a arquitetura: o padrão MVC (Model-View-Controller) [8], prevendo o uso futuro da interface gráfica, e conceitos da Arquitetura em Camadas [9], de modo que essa combinação foi vital para organizar claramente as responsabilidades do sistema.

No padrão MVC, o Model inclui classes como Cliente, Conta e Pessoa, contendo as regras de negócio. A View, feita com Flet [10], cuida da interface gráfica e navegação. O Controller gerencia solicitações da interface, validações e interação entre os Models e o segmento de manipulação de dados.

Na arquitetura em camadas, destacam-se o DAO (Data Access Object) [11], que simplifica operações leitura, atualização, criação e exclusão em JSON [12] e o Mapper [13], que centraliza a conversão entre JSON e objetos. Essas escolhas tornam o sistema modular e flexível, facilitando manutenção e futuras melhorias.

A camada de Modelos (Models), conforme dito anteriormente, foi utilizada para representar os elementos centrais do sistema, como clientes, contas (corrente e poupança) e pessoas. Nessas classes, além de armazenar os dados (nome, CPF, saldo e data de nascimento) foram definidos comportamentos específicos, como verificação e alteração de senha, cálculo de juros em conta poupança e limites de transferência em conta corrente. A classe Conta encapsula detalhes como tipo (depósito, saque, transferência), data e valor, permitindo gerar extratos e históricos. Para apoiar esses recursos, aplicou-se herança (contas especializadas derivando de uma conta genérica) e encapsulamento (acesso controlado a atributos sensíveis).

A camada de DAO (Data Access Object) trata do armazenamento e recuperação de dados em arquivos JSON. Sobre a classe genérica DAO foi construída uma hierarquia que inclui ClienteDAO, ContaDAO e PessoaDAO (todos herdando de DAO), cada uma responsável por criar, ler, atualizar e deletar registros da sua respectiva entidade (pessoa, conta e cliente). A separação estrita entre a lógica do programa e o armazenamento de dados permite mudar para outro formato de banco de dados apenas ajustando os DAOs, sem alterar o restante do sistema.

A camada de Mappers complementa os DAOs ao converter dicionários lidos do JSON em instâncias de objeto e vice-versa. Um ponto de destaque foi o uso de mapeadores específicos (PessoaMapper e ContaMapper) que buscam centralizar toda a lógica de desserialização e serialização, lidando com detalhes como formatação de datas, identificação de relacionamentos (ligando transações às contas correspondentes) e geração de identificadores únicos. Com isso, o código que consome essas entidades não precisa ”entender” sobre os formatos internos de armazenamento, mantendo-se limpo e focado em regras de negócio.

A camada de Controladores, vitais para a lógica da aplicação, que até então eram utilizados apenas expandiu-se para além do simples roteamento de chamadas: além do AuthController, que mantém sessões em memória para login e logout com acompanhamento de usuários ativos, há controladores dedicados para cliente, conta e transferências. Cada controlador recebe dados da interface, aplica validações (idade mínima, limites de transação, e regras de negócio), registra eventos em um logger estruturado e utiliza os modelos, DAOs e mapeadores para persistir (armazenar e manter) ou recuperar informações. Essa abordagem garante que todas as operações ocorram de forma coerente, com tratamento de erros padronizado e logs que facilitam auditoria e depuração.

A camada de View, implementada com o framework Flet, foi utilizada para fornecer uma interface gráfica interativa. As telas foram organizadas em módulos, sendo eles: tela para login, cadastro, abertura de conta, transferência e extrato. Também há um roteador de páginas que gerencia a navegação entre telas. Essa divisão também segue o padrão MVC (Model-View-Controller), mantendo a lógica de apresentação isolada da lógica de negócio e da persistência de dados.

Por fim, a camada de Utilitários (Utils) reuniu todas as funções auxiliares usadas ao longo do sistema. Entre elas estão validadores para CPF, e-mail, telefone e CEP (com integração à API ViaCEP [14] para preenchimento automático de endereços), formatação de datas e valores monetários, definição de constantes (como os caminhos de arquivos utilizados) e configuração de logger, responsável pela gravação de ocrorrências.

Centralizar essas funcionalidades evitou duplicação de código e tornou mais simples adicionar novas ferramentas (telas), funcionalidades de auxílio ou ajustar comportamentos de validação, seguindo a sugestão fornecida durante a avaliação inicial do código.

B. Principais Modulos/Classes

Conforme frisado até aqui, o sistema bancário desenvolvido adota uma arquitetura modular baseada nos principais conceitos da Programação Orientada a Objetos, proporcionando clareza, organização e facilidade de manutenção. Visando elucidar melhor as atribuições das principais classes presentes no sistema, serão detalhados algumas características da representação dessas classes. No núcleo do sistema estão as chamadas classes de modelo (Model), como Cliente, Conta e Pessoa. A classe Cliente representa cada usuário do banco, armazenando suas informações pessoais (através de um objeto Pessoa), sua senha e as contas bancárias (contas) às quais ele tem acesso. Isso facilita desde a autenticação até a realização de operações financeiras, uma vez que todas as informações relevantes ficam centralizadas e organizadas.As contas bancárias, por sua vez, são modeladas por uma classe abstrata chamada Conta, da qual derivam especializações como ContaCorrente e ContaPoupanca. Essa estrutura permite que diferentes tipos de conta compartilhem comportamentos comuns, como transferências e consultas de saldo, mas ainda possam apresentar regras específicas conforme o tipo de conta, como a aplicação de taxas ou rendimentos em suas respectivas atualizações mensais.

A classe Conta gerencia um histórico de operações (historico), onde cada evento financeiro é registrado como uma string descritiva contendo informações como data, valor e a

natureza da operação, permitindo assim a geração de extratos. Complementando esse conjunto, a classe Pessoa abstrai os dados pessoais dos clientes, diferenciando pessoas físicas (PessoaFisica) e jurídicas (PessoaJuridica), o que torna o sistema flexível e pronto para lidar com variados perfis de usuários.

As açoes sobre os módulos que armazenam os dados são coordenadas pelas classes controladoras (Controller). O AuthController, por exemplo, é o responsável por toda a lógica de autenticação, verificando credenciais de acesso e controlando a sessão do usuário (sessao-ativa) dentro do sistema. Já o CadastroController organiza o processo de cadastro de novos clientes, cuidando para que não haja duplicidade de documentos ou e-mails (utilizando ClienteDAO e PessoaDAO para as verificações), além de gerenciar a criação dos registros necessários tanto para a Pessoa quanto para o Cliente em si. O ContaController, por sua vez, centraliza diversas operações relacionadas às contas bancárias, como abertura de novas contas, consulta de extratos e encerramento. Essa separação dos controladores garante que cada tipo de operação seja tratado de maneira específica e independente, tornando o sistema mais seguro, confiável e simples de manter. O armazenamento e recuperação dos dados são realizados pelas classes DAO (Data Access Object) (DAO).

Cada tipo de dado principal (clientes, contas e pessoas) possui sua respectiva classe DAO (ClienteDAO, ContaDAO, PessoaDAO), responsável por gravar, buscar, atualizar e remover registros em arquivos no formato JSON. Dessa forma, toda a lógica de persistência(armazenar e manter) de dados fica isolada dessas outras camadas do sistema, o que facilita futuras alterações, como a adoção de outro formato de banco de dados. Além disso, todas as DAOs são baseadas em uma classe abstrata comum (DAO), que padroniza a forma de manipular os dados e garante que o sistema seja consistente independentemente do tipo de informação tratada. Para permitir que as informações transitem corretamente entre as diferentes partes do sistema, especialmente entre os objetos e sua representação em JSON, são utilizados mapeadores (Mapper). Eles são responsáveis, conforme já apresentado na seção de Arquiteturas, por converter os objetos das classes Conta e Pessoa (e suas subclasses) em dicionários apropriados para gravação nos arquivos, e vice-versa, através de classes como ContaMapper e PessoaMapper. Para a entidade Cliente, a lógica de mapeamento (conversão entre objeto e dicionário) é tratada internamente pela classe ClienteDAO. Assim, sempre que é preciso salvar ou recuperar uma conta ou uma pessoa, e de forma similar para um cliente, os dados podem ser facilmente transformados e utilizados pela aplicação sem perda de informação ou erros de interpretação.

O sistema também conta com diversas classes utilitárias (Utils), que complementam o funcionamento das demais camadas. Entre elas, destacam-se as classes de validação de dados (como ValidarCliente, ValidarConta, ValidarPessoa), que verificam se as informações digitadas pelos usuários, como senha ou número de conta, seguem os padrões estabelecidos. Outro exemplo importante é o uso de um logger para registrar os eventos relevantes e possíveis erros, o que facilita o acompanhamento do funcionamento do sistema e a identificação rápida de eventuais problemas. Além disso, há integrações externas, como a consulta de CEPs via API (através da classe API, presente em Utils), trazendo agilidade e precisão ao preenchimento dos dados de endereço dos usuários.

Por fim, toda a interação com o usuário acontece por meio da camada de interface gráfica (View), construída com o framework Flet. Essa camada é composta por diferentes telas, como a de login (TelaLogin), de cadastro (TelaCadastro) e o painel do usuário (TelaUsuario), além de diversos componentes visuais reutilizáveis, como campos de texto (campos.py), botões (botoes.py) e mensagens (mensagens.py). Um roteador organiza a navegação entre as telas, garantindo que o fluxo seja intuitivo e seguro, sempre levando o usuário ao lugar correto de acordo com as ações tomadas por ele mesmo.

Em resumo, a estrutura do sistema foi cuidadosamente pensada para que cada parte tivesse uma responsabilidade clara e bem definida. Assim, reafirmando o que foi apresentado na seção de Arquitetura, os modelos representam as entidades centrais, os controladores organizam a lógica das operações, os DAOs garantem o armazenamento seguro e eficiente dos dados, os mapeadores facilitam a conversão das informações, e as utilidades reforçam a segurança e a facilidade de uso. A interface gráfica, por sua vez, garante uma experiência amigável ao usuário final. Essa organização em camadas, aliada à aplicação dos princípios da orientação a objetos, torna o sistema robusto, escalável e fácil de entender, tanto para o usuário quanto para os desenvolvedores durante futuras manutenções ou expansões (algo que ocorreu com frequência durante a elaboração do projeto).

C. Aplicação da Programação Orientada a Objetos

Durante o desenvolvimento do sistema bancário, a Programação Orientada a Objetos (POO) foi essencial para deixar o código mais organizado, reutilizável e fácil de manter. Os quatro pilares da POO (abstração, encapsulamento, herança e polimorfismo) foram aplicados de forma prática para melhorar a estrutura e o funcionamento do sistema, a saber:

- **Abstração**: A abstração permitiu escolher quais atributos e métodos eram essenciais para representar uma determinada ”entidade”, permitindo dessa forma ocultar detalhes desnecessários para sua representação. Por exemplo, os dados pessoais no Cliente são gerenciados através de um objeto Pessoa associado, e a classe Cliente abstrai não apenas o acesso a alguns desses dados (como o numero de docuemto) mas também suas próprias responsabilidades, como o gerenciamento de senha e contas.Um outro exemplo é na classe ”Conta”, que abstrai operações financeiras genéricas mas sem expor diretamente como, por exemplo, o saldo é atualizado ou verificado.
- **Encapsulamento**: Em um sistema tão crítico como o escolhido pelo nosso grupo (envolve o uso de senhas, dados pessoais, movimentações monetárias), o encapsulamento é de vital importância. Ele permite proteger o estado de um objeto atravéz do uso de atributos privados/protegidos, expondo apenas métodos verificados

para acesso ou modificação de tais atributos. Atributos sensíveis como a senha de um cliente ou saldo de uma conta são marcados como privados, de modo que o acesso ou a modificação de tais atributos é realizado via métodos como ”validar-senha” e ”get-saldo”. Isso evita que outras partes do código alterem diretamente valores críticos presentes no sistema, garantindo consistência e integridade dos dados.

- **Herança**: A herança possibilita criar classes que herdam atributos e métodos de uma outra classe base (conhecida como pai, geralmente), podendo acrescentar ou modificar comportamentos. 2 classes evidentes que se utilizam de tal conceito são a classe ContaCorrente e a classe ContaPoupança, que herdam de Conta. Ambas as classes herdam características básicas como número, saldo e estado mas adicionam funcionalidades específicas (cálculo de juros para a poupança e taxa de manutenção para a corrente).
- **Polimorfismo**: O uso dele permitiu que diferentes classes fossem tratadas por uma interface comum, garantindo que cada uma responda de forma específica ao mesmo comando. Um exemplo dessa aplicação no seu código é o método ”atualizacao-mensal()” definido na classe abstrata Conta e implementado de formas diferentes nas subclasses ContaCorrente e ContaPoupanca. Aqui, de posse de uma lista de objetos Conta, ao chamar o método ”conta.atualizacao-mensal()” em cada conta, o método é executado dependendo se conta é uma ContaCorrente ou ContaPoupanca. A função ”transferir” na classe Conta também demonstra polimorfismo ao interagir com a ”conta-destino” que pode ser de qualquer subtipo de Conta.

Além dos exemplos relatados acima, há vários outros momentos em que os 4 Pilares da POO são aplicados ao longo do código. Além disso, há o constante uso e aplicação de outros conceitos que não foram abordados, como a Composição (associada a um relacionamento de ”parte-de”, como em PessoaDAO e ContaDAO com ClienteDAO) e a Agregação (associada ao relacionamento de ”tem-um”, como o cliente que pode ter uma ou mais contas).

D. Fluxo de Programa/Dados

O sistema final construído representa um produto resultante da integração entre as várias camadas de programa que foram descritas nas seções anteriores, conhecido como o ”back-end”, e o código desenvolvido para representação da interface com a qual o usuário tem contato, conhecida como ”front-end”. Essa integração é realizada através de fluxos de dados organizados em etapas sequenciais que garantem a coerência das operações e a segurança das informações dos usuários. A seguir, serão detalhados os fluxos principais, desde a autenticação até operações bancárias específicas, demonstrando como a interação entre interface, controladores, modelos e acesso a dados ocorre de maneira fluida e eficiente.

- **Fluxo de autenticação**: Ao abrir o aplicativo, o usuário encontra a TelaLogin, onde deve inserir seu documento de identificação (CPF ou CNPJ) e a senha, utilizando componentes como CampoCPF ou CampoCNPJ e CampoSenha. Ao confirmar a entrada, a aplicação ativa o processo de autenticação através da classe AuthController, responsável pela segurança dos acessos. Esse controlador primeiro verifica se as informações do usuário já estão salvas temporariamente em sua memória cache (-cache-clientes). Caso não estejam, o AuthController solicita ao ClienteDAO que consulte diretamente os arquivos que guardam os dados de clientes. Durante essa consulta, o ClienteDAO é a classe especializada em buscar e montar as informações completas dos clientes, o que pode envolver o uso do PessoaDAO e ContaDAO para obter dados pessoais e contas associadas. Para transformar os registros de arquivos JSON em objetos utilizáveis pelo sistema, o PessoaDAO e o ContaDAO contam com o auxílio do PessoaMapper e ContaMapper, respectivamente. Após o ClienteDAO localizar e construir o objeto Cliente, o AuthController utiliza este objeto para verificar se a senha informada corresponde à armazenada. Se os dados estiverem corretos, o Cliente é considerado logado, sendo sua sessão gerenciada pelo AuthController (em sessa-ativa), e o roteador de telas do sistema redireciona o usuário para sua página principal, com a TelaLogin exibindo uma mensagem de sucesso através do Notificador. Caso contrário, o Notificador da TelaLogin exibe um aviso de erro.
- **Fluxo de cadastro** :Na TelaLogin, o usuário tem a opção de criar uma nova conta, sendo encaminhado para a TelaCadastro. Nesta tela, deve fornecer seus dados pessoais completos (utilizando componentes como CampoNome, CampoEmail, etc.) e definir uma senha (CampoSenha). Dependendo do tipo de pessoa (física ou jurídica), campos específicos, como CampoDataNascimento ou nome-fantasia, são exibidos pela TelaCadastro. Ao confirmar o cadastro, a classe CadastroController verifica inicialmente se já existe um cliente cadastrado com o mesmo documento (consultando a classe ClienteDAO) ou e-mail (consultando a classe PessoaDAO). Se houver duplicidade, um erro será informado ao usuário através do Notificador na TelaCadastro. Caso não haja problemas, o CadastroController coordena a criação de um novo objeto Pessoa (PessoaFisica ou PessoaJuridica), utilizando o PessoaDAO (que internamente usa o PessoaMapper para a conversão de dados) para salvar essas informações em um arquivo JSON específico. Em seguida, o CadastroController gerencia a criação de um objeto Cliente, associado a aquela Pessoa (recuperada por meio da PessoaDAO) e à senha escolhida, e o ClienteDAO é utilizado para salvar este novo Cliente. Finalmente, o usuário recebe uma mensagem através do Notificador da TelaCadastro indicando sucesso no cadastro e é direcionado novamente à TelaLogin.
- **Fluxo de criação de conta**: Um usuário já autenticado, interagindo com a TelaUsuario, pode criar uma nova conta bancária na sua área pessoal. Ao escolher essa

opção, ele acessa a TelaCriarConta, onde seleciona o tipo de conta que deseja abrir (corrente ou poupança) através de um componente de seleção. Quando o usuário confirma a criação da conta, o ContaController verifica primeiro, com base nas informações do objeto Cliente (obtido via ClienteDAO), se este já possui uma conta daquele tipo. Caso não possua, o ContaController gera automaticamente um novo número de conta (após consultar o ContaDAO para listar contas existentes), cria o objeto Conta (ContaCorrente ou ContaPoupanca) correspondente ao tipo escolhido, e solicita ao ContaDAO (que utiliza o ContaMapper para a serialização) que salve essa nova conta nos registros JSON. Além disso, o ContaController atualiza os dados do Cliente (adicionando a nova conta à sua lista) e solicita ao ClienteDAO que persista essa atualização. Uma mensagem indicando sucesso ou erro, gerenciada pelo Notificador da TelaCriarConta, é exibida ao usuário.

- **Fluxo de Transferência**: Na TelaUsuario, é possível realizar transferências acessando a TelaPagamento. O usuário seleciona a conta que deseja utilizar para enviar dinheiro, insere o documento do destinatário (em CampoCPF ou CampoCNPJ), a conta destino (Dropdown, uma caixa de seleção que é preenchida após o PerfilController buscar os dados do destinatário), o valor (CampoValor) e a senha (TextField).Ao confirmar o pagamento, o PagamentoController realiza diversas validações: utiliza o ClienteDAO para buscar o cliente de origem e o objeto Cliente para verificar a senha, utiliza o ContaDAO para buscar a conta de origem e o objeto Conta para confirmar saldo suficiente e checar limites, e garante, através de sua lógica interna, que as contas envolvidas (objetos Conta de origem e destino, também obtidas via ContaDAO e ClienteDAO) sejam diferentes e ativas.Se todas as verificações forem bem-sucedidas, a transferência é efetuada internamente nos objetos Conta, atualizando os saldos e registrando essa operação no histórico de ambas as contas. As alterações são então salvas permanentemente através do ContaDAO, que atualiza os dois objetos Conta. O usuário recebe uma confirmação visual do sucesso ou falha da operação através do Notificador da TelaPagamento, e o saldo atualizado da conta é reexibido na tela.
- **Fluxo de Consulta - Extrato**: Dentro da TelaUsuario, o usuário pode consultar o extrato de suas contas acessando a TelaExtrato. Nela, escolhe a conta desejada a partir de uma lista preenchida com dados fornecidos pelo ContaController. Ao selecionar a conta, a TelaExtrato solicita ao ContaController que busque as informações detalhadas dela. O ContaController, por sua vez, utiliza o ContaDAO para recuperar o objeto Conta específico e verifica se está ativa. Caso esteja, o saldo atual e o histórico de transações são recuperados diretamente do objeto Conta. A TelaExtrato atualiza automaticamente sua interface exibindo o saldo (saldo-text) e a lista de transações (lista-extrato), utilizando componentes como CartaoTransacao para detalhar cada movimento financeiro, podendo este último recorrer ao ContaController para obter informações adicionais de outras partes envolvidas na transação, visando uma exibição mais completa.

E. Testes

Visando garantir a robustez do sistema, foram realizados testes na aplicação visando abranger os diferentes módulos que o compõem, assim como a integridade das regras de validação e persistência de dados.

No módulo Model, foram verificados a correta instanciação dos objetos principais, como PessoaFisica, PessoaJuridica, Cliente, ContaCorrente e ContaPoupanca, incluindo a integração com serviços externos simulados, validação de atributos, operações de transferência, atualização mensal e encerramento de contas, além da manipulação de exceções e representações textuais.

Já no módulo Utils e Validadores, testou-se a efetividade das regras de validação de dados como CPF, CNPJ, e-mail, saldo, número da conta e força de senha, garantindo a prevenção de dados inconsistentes ou malformados.

No módulo Mapper, assegurou-se a confiabilidade dos processos de serialização e desserialização dos objetos em dicionários JSON, validando tanto a leitura de dados externos quanto a persistência adequada das informações internas do sistema. Esse conjunto de testes buscou oferecer uma cobertura abrangente, validando as funcionalidades elementares do sistema, visto a importância delas para o correto funcionamento do mesmo. A instanciação errada de alguma classe do módulo central Model, por exemplo, pode prejudicar o correto funcionamento do programa, visto que é um módulo elementar, que é utilizado como base para os vários princípios de operações desenvolvidos ao longo do código. Para assegurar a qualidade e visualização dos testes, a estratégia adotada incluiu o uso da ferramenta Pytest [15], um framework de testes para Python que facilita a escrita de testes de pequena escala, mas que podem ser expandidos para suportar testes funcionais complexos. Sua importância no projeto reside na sua capacidade de automatizar a execução de testes, identificar falhas rapidamente e fornecer relatórios detalhados, o que é crucial para garantir que cada módulo, como Model, Utils, Validadores e Mapper, está funcionando conforme o esperado e que a integração entre eles esteja bem feita. Além disso, o pytest simplifica a escrita de e a organização dos testes, muito útil durante o desenvolvimento e execução de tais testes.

III. RESULTADOS

O desenvolvimento do Sistema de Gerenciamento Bancário culminou na entrega de uma aplicação funcional e didaticamente rica, que não apenas simula operações bancárias essenciais mas também exemplifica a aplicação de alguns princípios de engenharia de software. Nesse sentido, convém apresentar o produto final criado, com a apresentação de imagens que ilustram as sequências de telas e etapas com a qual o usuário se deparará, fazendo um paralelo com a

subseção ”Fluxo de Programa/Dados” - referente à seção ”Metodologias”.

Em um primeiro momento, como podemos ver na figura 1 o cliente se depara inicialmente com uma tela de login, caso não possua conta ele é convidado a criar uma.

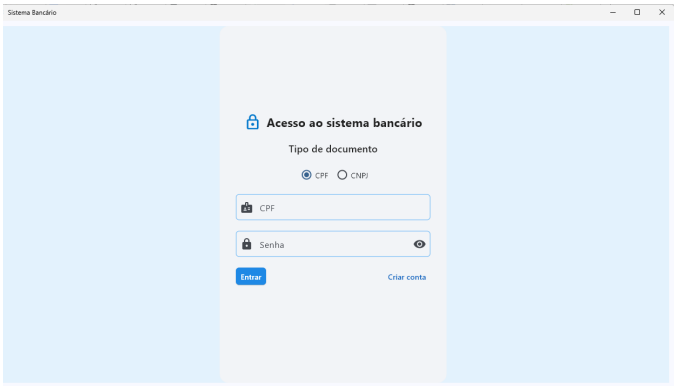

Fig. 1. Tela de login.

Impedimentos de login para usuários ainda não cadastrados, ou que estão inserindo senhas / CPF incorretos, limitam o acesso ao sistema, como pode ser visto na figura 2. Essa etapa é de vital importância para o funcionamento do sistema, pois permite a validação correta e robusta dos dados, além de garantir uma autenticação segura do usuário.

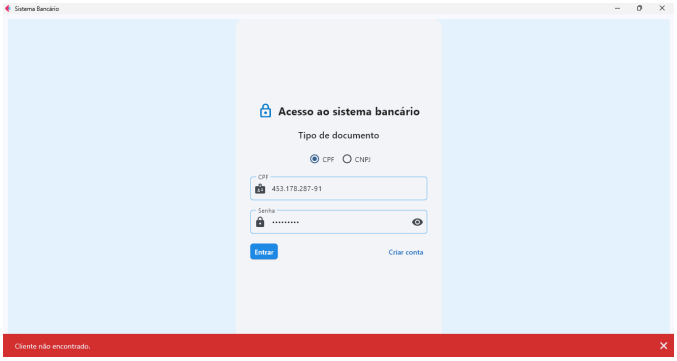

Fig. 2. Tentativa de login sem cadastro.

Em seguida, partimos para a tela de cadastro, por meio do uso do botão ”Criar conta”. Nela, são solicitados os dados necessários para a validação do cadastro do usuário, através do uso de uma interface amigável e intuivida que apresenta os padrões esperados para inserção nos respectivos campos. Aqui, também há o uso de recursos que impedem a inserção de informações errôneas, como CEP ou CPF errado, senha fora do padão, número de telefone com dígitos faltantes, etc. A figura 3 ilustra uma situação em que o usuário insere dados incorretos durante a tentativa de cadastro.

Tendo inserido informações corretas e compatíveis, o cadastro do usuário é finalizado, momento em que uma mensagem notificando a conclusão do cadastro é apresentada na tela. Também nesse momento, o usuário adquire o direito de acessar o sistema, com seu CPF e senha cadastrados.

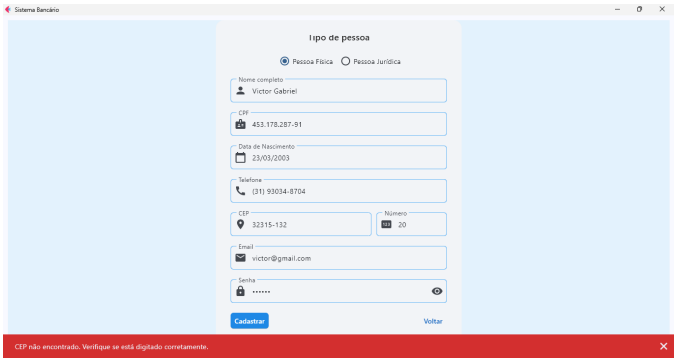

Fig. 3. Cadastro com dados inválidos

Aqui, as validações de senha e CPF continuam valendo, garantindo a segurança e robustez da aplicação. Conforme podemos ver na figura , o uso de uma senha diferente da que foi associada ao CPF durante o cadastro ”barra” a entrada do usuário que está tentando acessar o sistema. Verifica-se isso na figura 4, em que um usuário recentemente cadastrado tenta logar no sistema com uma senha diferente da cadastrada.

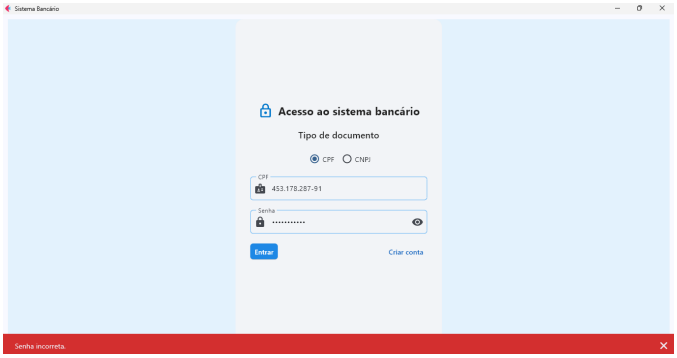

Fig. 4. Login com senha inválida

Inserindo senha e CPF corretos, o usuário tem seu acesso liberado para acessar o sistema. Aqui, em um momento inicial, ele tem contato com a aba ”Perfil” local em que constam os dados pessoais de cadastro do usuário, conforme observa-se na figura 5.

Tendo chegado até aqui, é hora do usuário criar uma conta, algo que pode ser feito na aba ”Criar conta”. A criação de uma conta permite ao usuário realizar, agora, operações bancárias como a visualização do extrato e transferências monetárias. Se a conta for devidamente criada, é retornada uma mensagem indicando o sucesso da operação. Tais processos podem ser observados nas figuras 6 e 7 .

Assim, as imagens a seguir ilustram as outras funcionalidades presentes no sistema, a saber : transferências, extrato, alteração de dados e gerenciamento de contas (envolvendo ativação e desativação de conta, tomando como base se ela está ativa ou não).

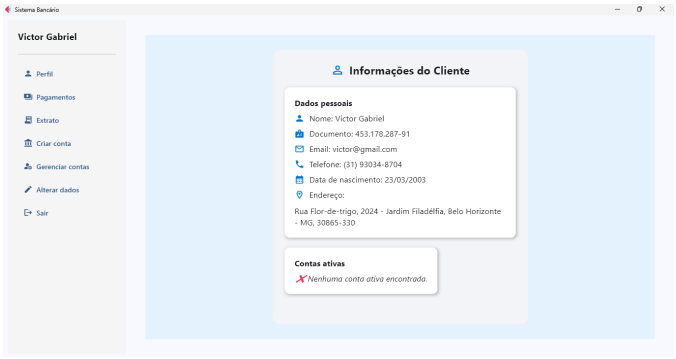

Fig. 5. Aba Perfil

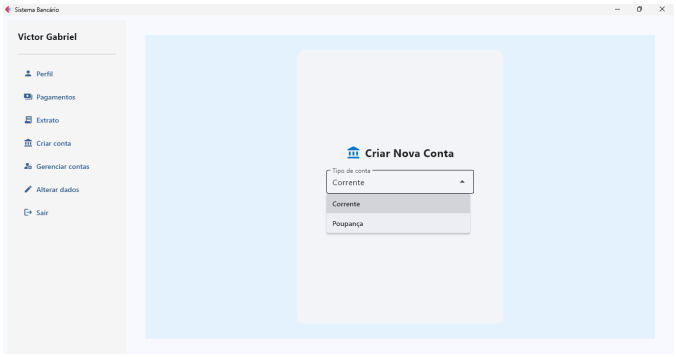

Fig. 6. Criação de conta

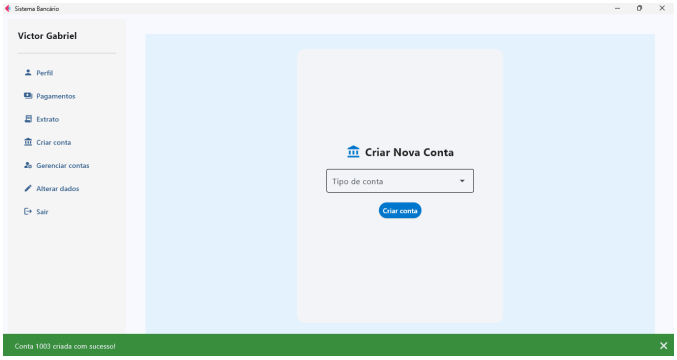

Fig. 7. Confirmação de criação de conta

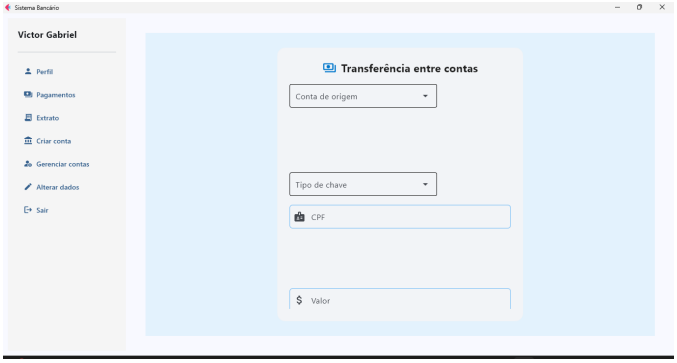

Fig. 8. Aba Pagamentos - pt.1

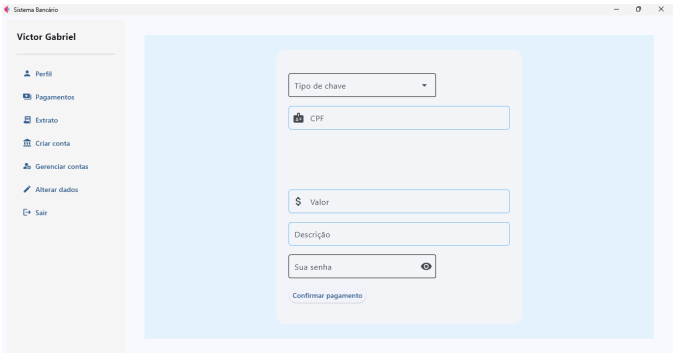

Fig. 9. Aba Pagamentos - pt2

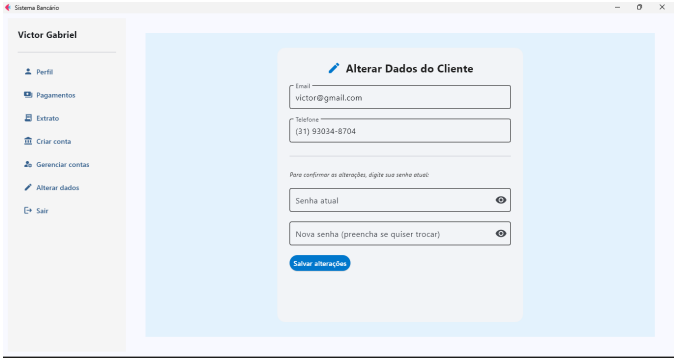

Fig. 10. Aba Alterar Dados

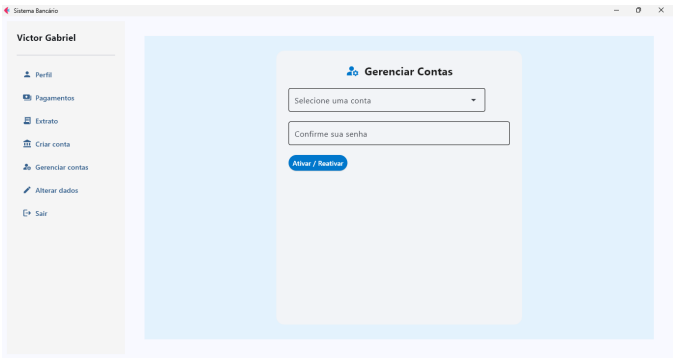

Fig. 11. Aba Gerenciar Contas

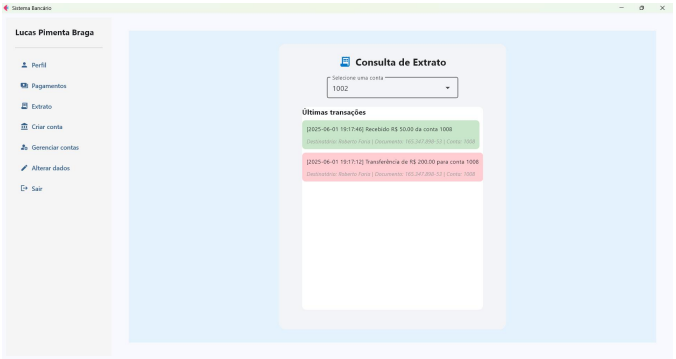

Fig. 12. Aba Extrato - Diferentes usuários

IV. Conclusão

O desenvolvimento do sistema bancário representou uma aplicação prática e abrangente dos conceitos vistos e relembrados em sala de aula, envolvendo desde a modelagem de entidades elementares como Cliente, Conta, PessoaFisica e PessoaJuridica até a construção de uma interface gráfica interativa com a biblioteca Flet. Utilizando os princípios da Programação Orientada a Objetos, o projeto aplicou de forma efetiva princípios como herança, polimorfismo, encapsulamento e abstração, enquanto a arquitetura modular organizada em camadas como Model, DAO, Mapper, Controller, View e Utils assegurou um código modular e de fácil entendimento, manutenção e expansão. Foram também implementados mecanismos de gerenciamento de dados, como a validação das informações com os validadores, a persistência através de arquivos JSON com o auxílio dos DAOs e a serialização e desserialização realizadas pelos Mappers. Além de simular processos bancários essenciais (autenticação, criação e gerenciamento de contas, transferências e pagamentos) o projeto consolidou uma ampla gama de aprendizados, como o domínio de padrões arquiteturais (MVC e DAO), a integração com APIs externas, o desenvolvimento de tratamento de exceções personalizadas, bem como o aproveitamento de funcionalidades características da linguagem Python, como as classes abstratas e métodos estáticos. Em suma, o sistema não apenas resultou em um software funcional e consistente, mas também serviu como uma sólida base de aprendizado prático, proporcionando experiência real em diversas etapas do ciclo de desenvolvimento de software.

References

[1] J. Silva and M. Pereira, "Evolução tecnológica dos serviços bancários," *Revista de Tecnologia Financeira*, 2023, [Online]. Available: https://www.researchgate.net/publication/355888668_EVOLUCAO_TECNOLOGICA_DOS_SERVICOS_BANCARIOS.

[2] G. F. Technologies, "A revolução digital no setor bancário brasileiro: Uma década de transformação," *Blog Galileo*, 2025, [Online]. Available: https://www.galileo-ft.com/pt/blog/revolucao-digital-setor-bancario-brasileiro-decada-transformacao/.

[3] DevMedia, "Os 4 pilares da programação orientada a objetos," *DevMedia*, 2022, [Online]. Available: https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264.

[4] TOTVS, "Front end: O que é, como funciona e qual a importância," 2021, acessado em: 22 maio 2025. [Online]. Available: https://www.totvs.com/blog/developers/front-end/

[5] L. Pimenta, M. Gontijo, and V. Silva, "Diagrama uml - sistema bancario," 2025. [Online]. Available: https://lucid.app/lucidchart/365f1fb1-cef3-475c-8bf7-16fc362b1218/edit?viewport_loc=3601%2C-7812%2C14463%2C6300%2C0_0&invitationId=inv_402e6509-3471-4268-836a-746181e79fa8

[6] "Sistema bancário," 2025. [Online]. Available: https://github.com/ateteu/sistema-bancario

[7] Alura. (2024, February) Padrões arquiteturais: arquitetura de software descomplicada. Acessado em: 26 de maio de 2025. [Online]. Available: https://www.alura.com.br/artigos/padroes-arquiteturais-arquitetura-software-descomplicada

[8] DevMedia, "Introdução ao padrão mvc," 2025, acessado em: 23 maio 2025. [Online]. Available: https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308

[9] DevMediaa, "Arquitetura de software: Introdução, camadas e concorrência," 2025, vídeo de apresentação. Acessado em: 23 maio 2025. [Online]. Available: https://www.devmedia.com.br/arquitetura-de-software-introducao-camadas-e-concorrencia/26124

[10] D. do Flet. (2025) Flet: Construa aplicativos multiplataforma com python e flutter. Acesso em: 22 maio 2025. [Online]. Available: https://flet.dev/

[11] Stack Overflow, "Como funciona o padrão dao?" 2025, acessado em: 22 maio 2025. [Online]. Available: https://pt.stackoverflow.com/questions/113840/como-funciona-o-padr%C3%A3o-dao

[12] Oracle, "O que é json?" 2025, acessado em: 23 maio 2025. [Online]. Available: https://www.oracle.com/br/database/what-is-json/

[13] DevMedia, "Qual a diferença de um dao para um mapper?" 2025, acessado em: 23 maio 2025. [Online]. Available: https://www.devmedia.com.br/forum/qual-a-diferenca-de-um-dao-para-um-mapper/583417

[14] DevPonto, "Consulta de cep com python – via api de cep," *DevPonto*, 2024. [Online]. Available: https://www.devponto.com/posts/consulta-de-cep-com-python-buscando-enderecos/

[15] A. Q. Assurance, "Tutorial de pytest para iniciantes," 2025, acessado em: 27 de junho de 2025. [Online]. Available: https://medium.com/assertqualityassurance/tutorial-de-pytest-para-iniciantes-cbdd81c6d761