RELATÓRIO DE REFACTORING E CLEAN CODE

SISTEMA BANCÁRIO

Disciplina:Gestão e Qualidade de Software
Professor(a): Robson Calvetti
Integrantes:

- Gabriel Fernandes Bispo RA: 823143267
- João Luiz da Silva RA: 82420546
- Pedro Henrique Braz Lira RA: 823117291
- Pedro Zorzete Brisolla de Campos RA: 823147399

Data: Novembro/2025

Introdução

Este relatório apresenta o trabalho de refatoração, melhoria de qualidade de código e criação de testes unitários realizados no projeto A3 do Sistema Bancário. O objetivo é evidenciar as deficiências do código original, justificar as alterações implementadas, descrever os testes unitários adicionados e, por fim, discutir a importância dos princípios de Clean Code na manutenção de software.

O repositório utilizado pelo grupo é público e está disponível no GitHub no seguinte endereço: https://github.com/Projeto-A3-GQS/A3-Sistema-Bancario

I. Descrição das deficiências do código original

O código original do sistema bancário foi pegado de uma antiga atividade feita por um dos integrantes no passado. Essa versão inicial, armazenada no mesmo repositório em seu estado mais antigo, apresentava uma série de limitações de arquitetura, organização e manutenibilidade.

De forma geral, o projeto consistia em um único script concentrando regras de negócio, definição de classes, controle de fluxo e interface de linha de comando. Essa abordagem dificulta a evolução do sistema, a realização de testes automáticos e o trabalho colaborativo entre os membros do grupo.

Principais deficiências identificadas

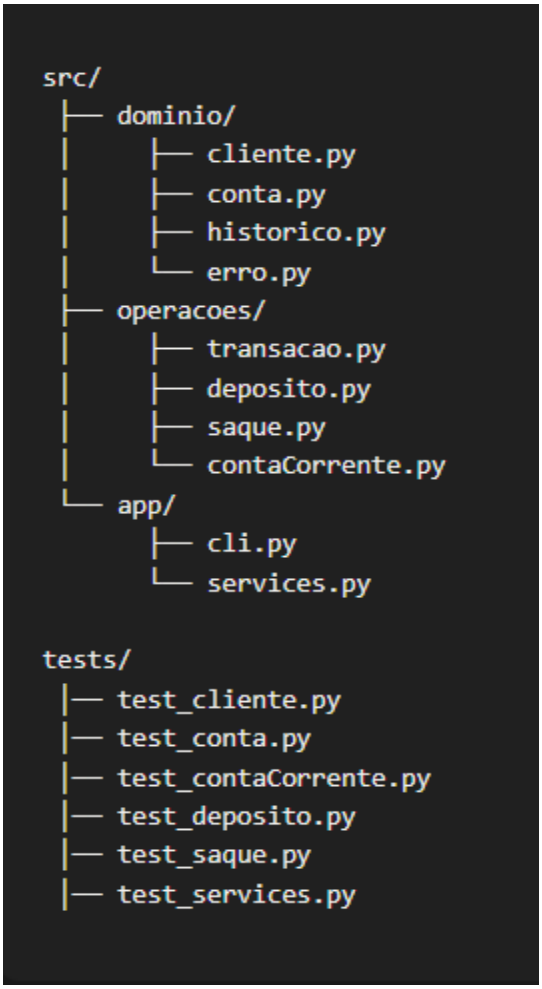
- Estrutura monolítica em um único arquivo de código, reunindo classes, funções, menu e interação com o usuário em um mesmo lugar, o que gerava forte acoplamento e baixa coesão.
- Ausência de modularização em pacotes, impedindo a separação clara entre camada de domínio, operações bancárias e interface de usuário.
- Falta de encapsulamento adequado, com atributos expostos de forma direta e pouca utilização de propriedades para proteger o estado interno das entidades de domínio.
- Mistura de regras de negócio com detalhes de entrada e saída, como chamadas diretas a funções de input e impressão no console, o que tornava o código difícil de testar e de reutilizar.
- Pouca padronização de nomenclatura e formatação, com mensagens inconsistentes, estilo de código heterogêneo e ausência de documentação mínima.
- Inexistência de testes automatizados, o que impossibilitava validar de forma sistemática regras essenciais do sistema, como limites de saque, registro de histórico e vínculos entre clientes e contas.

II. Justificativas para as mudanças feitas no código

A refatoração do código foi conduzida com o objetivo de transformar um sistema originalmente monolítico, difícil de manter e pouco estruturado em um software modular, claro e alinhado aos princípios de Clean Code. Todas as alterações foram devidamente registradas na branch *develop*, e cada decisão tomada teve como prioridade a melhoria da legibilidade, da escalabilidade e da organização interna do projeto.

2.1 Reorganização estrutural do projeto

A primeira etapa da refatoração consistiu na reorganização completa da estrutura do projeto. O código original encontrava-se centralizado em um único arquivo, o que dificultava sua manutenção. Para corrigir isso, a equipe dividiu o projeto em módulos específicos dentro de src/, resultando na seguinte estrutura:



O módulo **dominio** passou a concentrar as entidades centrais, como Cliente, Conta, Histórico e Erros. Já o módulo **operacoes** ficou responsável pela modelagem das transações bancárias, incluindo a classe abstrata Transacao, as operações de saque e depósito e a implementação específica da ContaCorrente. Por fim, o módulo **app**, agora contendo `cli.py` e `services.py`, passou a representar a camada de interface e orquestração do sistema. Essa separação tornou o código mais claro, eliminou o acoplamento excessivo entre camadas e criou uma arquitetura sustentável.

2.2 Melhoria e encapsulamento das entidades de domínio

Com a organização estrutural definida, foi possível melhorar diretamente as classes do domínio. Os atributos que antes eram públicos passaram a ser privados, com acesso controlado por meio de propriedades. Esse movimento aumentou a segurança do estado interno de cada entidade, minimizando a chance de inconsistências e comportamentos inesperados. Além disso, métodos extensos foram reescritos de maneira mais enxuta, reforçando o princípio de responsabilidade única.

Outro ponto fundamental foi o uso da classe **RegraDeNegocioError**, que centraliza o tratamento de erros relacionados ao negócio, tornando mais claras as situações que violam as regras da aplicação.

2.3 Extração e organização das operações bancárias

No código original, as operações de saque e depósito estavam misturadas com a lógica da Conta e até com a interface. Durante a refatoração, essas regras foram movidas para o módulo operações/, onde passaram a existir como classes independentes. A introdução de uma classe abstrata Transacao permitiu definir um contrato comum para todas as operações financeiras, garantindo padronização e abrindo portas para futuras implementações, como transferência entre contas.

A ContaCorrente também foi separada e ganhou seu próprio arquivo dentro desse módulo, tornando mais claras suas regras adicionais, como limites e quantidade diária de operações. Isso deixou o sistema mais modular e coerente, além de facilitar a escrita de testes unitários específicos para cada tipo de operação.

2.4 Separação da interface e criação do módulo app/

Uma das principais fontes de confusão no código original era a mistura entre interface e lógica de negócio. Para resolver esse problema, foi criado o módulo **app**, dividido em dois arquivos:

- **cli.py**, responsável exclusivamente pela interação com o usuário (menus, entradas e saídas).
- **services.py**, responsável pela orquestração geral do sistema, conexão entre domínio e operações e fluxo de execução.

Essa separação permitiu que a interface fosse tratada de forma independente das regras de negócio, tornando o sistema mais flexível e conforme às boas práticas de arquitetura.

2.5 Documentação centralizada no README

Em vez de inserir comentários extensos dentro do código, o grupo optou por concentrar a documentação no arquivo **README.md**. Esse documento passou a descrever toda a estrutura do projeto, tecnologias utilizadas, fluxo de execução, arquitetura, testes e instruções de uso. A decisão segue a abordagem moderna de evitar poluição do código-fonte com comentários desnecessários, mantendo tudo o que diz respeito ao entendimento geral do sistema em um local único, organizado e acessível.

2.6 Padronização e harmonização geral do projeto

Após a reorganização e criação dos módulos, foi realizada uma etapa de padronização: ajustes de nomenclatura, correção de importações inconsistentes, eliminação de arquivos duplicados, reorganização da formatação e simplificação de trechos redundantes. Essa fase foi essencial para garantir que todo o código seguisse uma linha uniforme, facilitando tanto a leitura quanto a colaboração entre os membros da equipe.

2.7 Commits e Período de Refatoração

Todas as mudanças descritas foram registradas por meio de commits realizados no branch develop do repositório público entre o início de outubro e o final de novembro de 2025, atendendo ao requisito de que cada integrante do grupo realizasse pelo menos um commit em seu próprio nome. Dessa forma, o histórico

do Git documenta a evolução do projeto desde o código original até a versão refatorada.

O versionamento foi feito pensando em como funciona o ambiente de trabalho, pensando sempre em criar branchs a partir da develop para subir novos códigos e só quando testados e funcionado que aí sim a branch develop foi mergeada na main.

Assim os commits dos integrantes estão feitos na branch da develop, seguindo o padrão do GitFlow.

III. Descrição dos testes unitários implementados

Um dos objetivos centrais do trabalho foi introduzir testes unitários no projeto. Para isso, foi criada a pasta tests na raiz do repositório e configurado o uso do framework pytest por meio do arquivo pytest.ini. Os testes têm como função validar o comportamento das principais classes de domínio e operações bancárias.

3.1 Estrutura da pasta de testes

A pasta de testes segue a convenção do pytest, utilizando nomes de arquivos iniciados com o prefixo test_. Cada arquivo de teste agrupa casos de teste relacionados a um conjunto específico de funcionalidades do sistema, como contas, clientes ou operações de saque e depósito. Essa organização facilita a leitura e o entendimento da cobertura de testes.

3.2 Testes de entidades de domínio

Os testes criados para as entidades de domínio concentram-se em validar a correta criação e o comportamento de classes como Cliente, Pessoa Física e Conta. Entre os pontos testados, destacam-se:

- Criação de clientes com dados válidos, garantindo que os atributos obrigatórios sejam armazenados de forma correta.
- Associação de contas a clientes, verificando se uma conta é adicionada apenas uma vez e se o vínculo é mantido de forma consistente.
- Uso de propriedades para leitura de dados de cliente, evitando mudanças indevidas após a criação do objeto.

3.3 Testes de contas e histórico

As classes responsáveis por contas bancárias e histórico de operações também foram alvo de testes. Entre as validações realizadas, estão:

- Depósitos com valores válidos, garantindo o aumento correto do saldo e a inclusão da operação no histórico.
- Saques respeitando o saldo disponível e, no caso de contas correntes, os limites adicionais configurados.
- Registro adequado das operações no histórico, incluindo data, tipo de transação e valor, permitindo auditoria das movimentações.

3.4 Testes das operações de saque e depósito

As classes que representam operações específicas, como saque e depósito, foram testadas para assegurar que a lógica de cada transação respeita as regras definidas pelo domínio. Os testes verificam, por exemplo, se uma transação só é registrada no histórico quando a operação é concluída com sucesso.

3.5 Benefícios obtidos com a inclusão de testes

A introdução de testes unitários trouxe benefícios diretos ao projeto. Sempre que uma refatoração foi realizada, os testes puderam ser executados para garantir que o comportamento esperado se mantivesse. Isso reduziu o risco de regressões e aumentou a confiança do grupo em relação às mudanças de código.

Além disso, os testes servem como documentação executável das regras de negócio, descrevendo em forma de código exemplos concretos de uso do sistema.

IV. Conclusão sobre a importância do Clean Code

O trabalho realizado no projeto da A3 ilustra, de forma prática, a importância dos princípios de Clean Code na manutenção e evolução de sistemas de software.

A partir de um código inicial concentrado em um único arquivo, sem testes e com responsabilidades misturadas, foi possível, por meio de refatorações graduais, alcançar uma arquitetura mais modular, legível e testável. A separação em pacotes de domínio, operações e interface reduziu o acoplamento entre componentes, facilitando a compreensão do sistema e a colaboração entre os membros do grupo.

O uso de encapsulamento, tipagem explícita e padronização de nomes contribuiu para a clareza do código, diminuindo ambiguidades e tornando mais difíceis os erros de uso incorreto das classes. Já a inclusão de testes unitários

criou uma rede de segurança que permite refatorar e evoluir o sistema com mais confiança.

De maneira geral, o projeto demonstra que Clean Code não é apenas uma questão estética, mas um investimento na qualidade e na longevidade do software. Códigos limpos são mais fáceis de entender, de corrigir e de expandir, trazendo benefícios tanto para os desenvolvedores quanto para os usuários finais.