RELATÓRIO DE REFACTORING E CLEAN CODE

SISTEMA BANCÁRIO

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

- Gabriel Fernandes Bispo 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 ou comentários explicativos.
- 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.

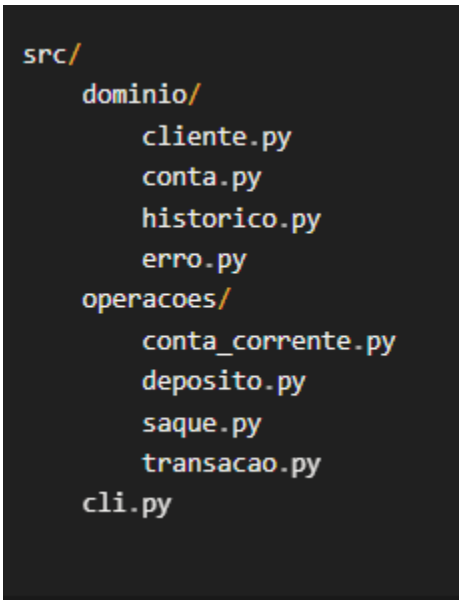
II. Justificativas para as mudanças feitas no código

A partir da análise das deficiências do código original, o grupo planejou e executou uma série de refatorações organizadas em commits no branch develop do repositório, com datas a partir de outubro de 2025. Essas mudanças seguiram princípios de Clean Code e boas práticas de projeto orientado a objetos.
Os principais objetivos da refatoração foram:

- Separar responsabilidades em módulos e pacotes específicos, evitando o acoplamento excessivo do arquivo original.
- Isolar o domínio de negócio da camada de interface, permitindo evolução futura do sistema com outras interfaces sem alterar as regras centrais.
- Melhorar a legibilidade e a consistência do código, por meio de padronização de nomes, formatação e uso de tipagem estática.
- Criar uma base adequada para implementação de testes unitários, tornando o sistema mais confiável e fácil de manter.

2.1 Modularização da estrutura do projeto

Uma das mudanças mais importantes foi a reorganização estrutural do projeto. O código foi movido para a pasta src, e o sistema passou a utilizar uma divisão em pacotes, com uma organização semelhante à estrutura abaixo:



Essa organização torna explícitas as responsabilidades de cada parte do sistema. O pacote domínio concentra as entidades centrais, como Cliente, Conta e Histórico. O pacote operações reúne as operações bancárias, como saque e depósito, enquanto o arquivo de interface (cli.py) se responsabiliza apenas pela interação com o usuário na linha de comando.

2.2 Encapsulamento e melhoria das classes de domínio

As classes de domínio passaram por refatorações importantes, com foco em encapsulamento, clareza e segurança de dados. Atributos passaram a ser declarados como privados e expostos através de propriedades, evitando o acesso direto e descontrolado ao estado interno. Além disso, foram introduzidos tipos explícitos nos métodos e construtores, melhorando a legibilidade e auxiliando ferramentas de análise estática.

Também foi criada uma exceção específica para regras de negócio, permitindo sinalizar violações de forma mais controlada e expressiva, em vez de depender de códigos de retorno ou mensagens genéricas.

2.3 Refatoração das operações bancárias

As operações bancárias, como saque e depósito, foram extraídas do código original e reorganizadas em classes especializadas, derivadas de uma classe abstrata de transação. Essa abordagem trouxe diversos benefícios:

- Definição de um contrato comum para diferentes tipos de transações, facilitando a extensão futura com novas operações, como transferência ou pagamento de contas.
- Isolamento da lógica de cada operação em sua própria classe, com responsabilidade clara por validar valores, interagir com a conta e registrar o resultado no histórico.
- Redução do acoplamento da camada de interface com os detalhes da implementação das regras bancárias.

Outro ponto importante foi o ajuste no registro de histórico. O sistema passou a registrar as transações somente quando a operação é bem-sucedida, garantindo que o histórico reflita fielmente o estado da conta.

2.4 Separação clara entre domínio e interface

Uma das fragilidades do código original era a mistura de regras de negócio com a interface de linha de comando. A refatoração isolou o menu de opções, a leitura de entradas e a exibição de mensagens em um módulo específico de interface, deixando o domínio independente de detalhes de apresentação.

Com isso, fica mais simples evoluir o sistema para utilizar outra interface, como uma API web ou uma interface gráfica, reaproveitando o mesmo núcleo de domínio e operações bancárias.

2.5 Padronização e melhoria da legibilidade

Durante o processo de refatoração, o grupo também padronizou nomes de variáveis, métodos e classes, corrigiu mensagens exibidas ao usuário e organizou melhor o fluxo de leitura do código. Espaços em branco desnecessários foram removidos e trechos redundantes foram simplificados, contribuindo para um código mais limpo e profissional.

2.6 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 no ambiente de trabalho, pensando sempre em criar branchs a partir da develop para subir

novos códigos e só quando testados e funcionando 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.