---

**i. Descrição das deficiências do código identificado**

**Repositório GitHub (versão inicial - até 30/09/2025):**

🧠 **Código analisado:**

**Arquivo:** BankApp.java
**Descrição:** Aplicação de simulação bancária com interface Swing, representando um sistema bancário simples.

⚙️ **Principais deficiências encontradas:**

| Tipo de Problema | Descrição Detalhada | Impacto |
|---|---|---|
| **1. Nomes genéricos** | Variáveis como a, b, c, d, h, i tornam o código ilegível. | Dificulta a leitura e manutenção. |
| **2. Falta de modularização** | Toda a lógica e interface estão na mesma classe BankApp. | Viola o **Princípio da Responsabilidade Única (SRP)**. |
| **3. Ausência de orientação a objetos** | O sistema não possui classes como Cliente e ContaBancaria. | Impossibilita reaproveitamento de código. |
| **4. Falta de validações** | Permite depósito negativo e criação de conta sem nome. | Gera resultados incorretos e instabilidade. |
| **5. Tratamento de erros genérico** | Mensagens como “Erro no depósito” não explicam o problema. | Dificulta depuração e confunde o usuário. |
| **6. Interface rígida** | Usa setLayout(null) e posições fixas. | Dificulta adaptação para diferentes resoluções. |
| **7. Falta de testes** | Nenhuma estrutura automatizada de verificação. | Reduz a confiabilidade do sistema. |

Essas deficiências caracterizam o código como **“código sujo”**, por violar boas práticas de clareza, modularidade e manutenibilidade.

---

**ii. Justificativas para as mudanças feitas no código refatorado**

**Repositório GitHub (versão refatorada - commits de 01/10/2025 a 27/11/2025):**

### 🧩 Estrutura do código refatorado

O código refatorado foi reorganizado para aplicar **princípios de Clean Code e Programação Orientada a Objetos (POO)**.
A estrutura do novo sistema ficou assim:

- BancoAppGUI.java → Interface gráfica (Swing)
- ContaBancaria.java → Lógica de negócio (depósito, saque, saldo)
- Cliente.java → Representa o titular da conta

### 🔄 Mudanças e justificativas

| Mudança | Justificativa Técnica | Benefício |
|---|---|---|
| **Criação de classes Cliente e ContaBancaria** | Separação da lógica de negócio da interface gráfica. | Segue o **encapsulamento** e o **SRP**. |
| **Nomes claros** (nomeField, valorField, saidaArea) | Melhora a legibilidade e autoexplicação do código. | Facilita o trabalho em equipe. |
| **Métodos específicos** (criarConta(), depositar(), sacar(), mostrarSaldo()) | Evita repetição e centraliza a lógica. | Código mais limpo e modular. |
| **Validações robustas** | Impede operações inválidas (como saque sem saldo). | Evita erros e mantém integridade. |
| **Mensagens personalizadas** | Substitui mensagens genéricas por feedbacks explicativos. | Melhora a experiência do usuário. |
| **Tratamento de exceções refinado** | Uso de IllegalArgumentException com mensagens específicas. | Torna o código mais previsível. |
| **Preparação para testes unitários** | Separação das classes facilita uso de JUnit. | Aumenta a confiabilidade do sistema. |

📌 Cada aluno do grupo deverá realizar ao menos **um commit** no repositório entre 01/10 e 27/11/2025, documentando claramente sua contribuição.

---

**iii. Descrição dos testes unitários implementados**

**Pasta de testes:**

📁 /testes

**Link:**

**Exemplo de teste implementado (ContaBancariaTest.java):**

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;


public class ContaBancariaTest {


  @Test

  public void testDepositoValido() {

    ContaBancaria conta = new ContaBancaria(new Cliente("Correa"));

    conta.depositar(100);

    assertEquals(100, conta.getSaldo());

  }


  @Test

  public void testSaqueInvalido() {

    ContaBancaria conta = new ContaBancaria(new Cliente("Correa"));

    conta.depositar(50);

    Exception ex = assertThrows(IllegalArgumentException.class, () ->
conta.sacar(100));

    assertEquals("Saldo insuficiente!", ex.getMessage());

  }


  @Test

  public void testDepositoNegativo() {
```

```
        ContaBancaria conta = new ContaBancaria(new Cliente("Correa"));
        Exception ex = assertThrows(IllegalArgumentException.class, () -> conta.depositar(-10));
        assertEquals("Valor inválido para depósito!", ex.getMessage());
    }
}
```

🧩 **O que os testes verificam:**

- Depósitos válidos aumentam o saldo corretamente.
- Saques maiores que o saldo geram exceção.
- Depósitos negativos são bloqueados.
- O saldo se mantém consistente após cada operação.

✅ **Ferramenta utilizada:** JUnit 5
✅ **Objetivo:** garantir integridade e confiabilidade no comportamento das classes refatoradas.

---

**iv. Conclusão – A importância do Clean Code na manutenção de software**

A refatoração do código **Banco Ruim v1** para **Banco Limpo v2** demonstra que o **Clean Code é essencial para o ciclo de vida saudável de um software**.
Mais do que deixar o código bonito, as práticas de Clean Code:

- reduzem o tempo de manutenção;
- facilitam o trabalho em equipe;
- evitam erros e comportamentos inesperados;
- e permitem evolução contínua com segurança.

Um sistema com código limpo é **compreensível, previsível e confiável**.
A implementação dos princípios de **simplicidade (KISS)**, **não repetição (DRY)** e **responsabilidade única (SRP)** mostra que **bons padrões de código garantem longevidade ao sistema e eficiência ao desenvolvedor**.

O projeto **Banco Limpo v2** reflete uma transformação técnica e conceitual:
de um código funcional, mas confuso, para um sistema limpo, modular e sustentável — provando que **refatorar é investir na qualidade do software e na produtividade da equipe.**

---

## ✍️ Autores

- **Correa (responsável pela refatoração principal e criação dos testes)**
- (Adicionar os demais integrantes do grupo, caso existam)