Relatório Completo: Refatoração de Código - Cálculo de Porcentagem

1. Código Original e suas Deficiências Identificadas

O código original consistia em duas funções p1 e p2 que realizavam exatamente a mesma operação: cálculo da porcentagem. Ambas recebiam dois parâmetros e realizavam a divisão da parte pelo total, multiplicando o resultado por 100. Embora funcional, o código apresentava diversas falhas graves segundo as boas práticas de programação:

- **Código duplicado:** As funções p1 e p2 tinham a mesma lógica, o que é um claro exemplo de violação do princípio DRY (Don't Repeat Yourself).
- **Nomes ruins:** Os nomes p1 e p2 não são descritivos e dificultam o entendimento da funcionalidade do código.
- **Ausência de validação:** Não havia tratamento para divisão por zero, o que poderia causar erros em tempo de execução.
- **Não orientado a objetos:** O código seguia uma abordagem puramente procedural, dificultando a manutenção e expansão futura.
- **Baixa legibilidade e extensibilidade:** Caso fosse necessário implementar novos tipos de cálculo de porcentagem, seria preciso criar novas funções repetitivas.

Exemplo do código original:

```python
def p1(x, y):
    return (x / y) * 100

def p2(x, y):
    return (x / y) * 100

print("Porcentagem 1:", p1(50, 200))
```

```
print("Porcentagem 2:", p2(30, 150))
```

2. Código Refatorado e Justificativas para as Mudanças Feitas

O código foi completamente refatorado para adotar a programação orientada a objetos (OOP), além de implementar o Design Pattern Strategy. Com isso, conseguimos modularizar a lógica de cálculo da porcentagem, tornando o sistema mais flexível e robusto.

Principais mudanças e justificativas:

- **Orientação a objetos:** Criamos uma interface (classe abstrata) chamada CalculadoraPorcentagem que define o método calcular. Dessa forma, é possível criar diversas implementações específicas de cálculo.
- **Design Pattern Strategy:** Implementamos duas classes concretas: PorcentagemSimples e PorcentagemComArredondamento, cada uma representando uma estratégia distinta de cálculo de porcentagem. Isso segue o princípio Open/Closed, permitindo adicionar novos métodos de cálculo sem modificar o código existente.
- **Validação de dados:** Adicionamos tratamento para evitar divisão por zero, lançando uma exceção caso o total seja igual a zero.
- **Legibilidade e extensibilidade:** Os nomes das classes e métodos são descritivos, facilitando o entendimento e manutenção.

Exemplo do código refatorado:

```python
from abc import ABC, abstractmethod

class CalculadoraPorcentagem(ABC):
    @abstractmethod
    def calcular(self, parte, total):
        pass
```

```
class PorcentagemSimples(CalculadoraPorcentagem):
    def calcular(self, parte, total):
        if total == 0:
            raise ValueError("O total não pode ser zero.")
        return (parte / total) * 100

class PorcentagemComArredondamento(CalculadoraPorcentagem):
    def calcular(self, parte, total):
        if total == 0:
            raise ValueError("O total não pode ser zero.")
        return round((parte / total) * 100, 2)
```

3. Testes Unitários Implementados

Após a refatoração, foi fundamental garantir que as novas implementações funcionassem corretamente. Para isso, criamos testes unitários utilizando o módulo unittest do Python. Os testes verificam:

- O cálculo correto da porcentagem simples.
- O cálculo correto da porcentagem com arredondamento.
- O lançamento de exceção no caso de divisão por zero.

Exemplo de testes implementados:

```python
import unittest
from codigo_refatorado import PorcentagemSimples, PorcentagemComArredondamento

class TestPorcentagem(unittest.TestCase):
    def test_porcentagem_simples(self):
        p = PorcentagemSimples()
        self.assertEqual(p.calcular(50, 200), 25.0)
```

```
    def test_porcentagem_arredondada(self):
        p = PorcentagemComArredondamento()
        self.assertEqual(p.calcular(30, 150), 20.0)

    def test_divisao_por_zero(self):
        p = PorcentagemSimples()
        with self.assertRaises(ValueError):
            p.calcular(10, 0)

if __name__ == "__main__":
    unittest.main()
```

4. Conclusão sobre a Importância do Clean Code na Manutenção de Software

O processo de refatoração deste código demonstrou, na prática, os benefícios de se adotar boas práticas de desenvolvimento e princípios de Clean Code. A versão inicial, apesar de funcionar, era limitada, propensa a erros e difícil de manter.

Com a aplicação da programação orientada a objetos e de Design Patterns, o código se tornou mais organizado, legível e preparado para futuras expansões. A validação de entrada garante maior robustez e segurança, enquanto os testes unitários fornecem uma base sólida para identificar rapidamente qualquer regressão.

Em ambientes profissionais, onde sistemas evoluem constantemente, práticas como essas são essenciais para garantir a qualidade do software, reduzir custos de manutenção e facilitar o trabalho em equipe.

Lucas Urbanski dos Santos
GitHub: https://github.com/zLucaszzz/projeto_porcentagem