Análise e Refatoração de Código Legado - Clean Code

Natan Vigil de Azambuja - 12923113102
Nícholas Thomas Nagel Ivã de Almeida - 12723115874

1. Código Original e Deficiências Identificadas

Código Original:

```
python
import math

def calcular_area(tipo, a, b=0):
 if tipo == 'quadrado':
 return a * a
 elif tipo == 'retangulo':
 return a * b
 elif tipo == 'circulo':
 return math.pi * a * a
 else:
 return None

print(calcular_area('quadrado', 4))
print(calcular_area('retangulo', 4, 5))
print(calcular_area('circulo', 3))
print(calcular_area('triangulo', 4, 5))
```

Problemas Identificados:

* Violação do Princípio Open/Closed (SOLID): Para adicionar novas formas, é necessário modificar a função existente.

* Nomes pouco descritivos: Parâmetros a e b não são autoexplicativos. * Estrutura

condicional complexa: Muitos if/elif tornam o código difícil de manter. * Retorno

inconsistente: Retorna None para tipos não reconhecidos, o que pode causar erros. * Falta

de tipagem: Não há indicação dos tipos esperados para os parâmetros. * Acoplamento alto:

A função faz tudo, violando o Single Responsibility Principle. * Falta de tratamento de erros:

Não valida entradas negativas, por exemplo.

2. Código Refatorado

Implementação com Padrão Strategy:

```
python
from abc import ABC, abstractmethod
import math
from typing import Protocol

class Forma(Protocol):
 @property
 def area(self) -> float:
 """Calcula a área da forma geométrica"""  ...

class Quadrado:
 def _init_(self, lado: float):
 if lado <= 0:
 raise ValueError("Lado deve ser positivo")
self.lado = lado

 @property
 def area(self) -> float:
 return self.lado ** 2

class Retangulo:
 def _init_(self, base: float, altura: float):
 if base <= 0 or altura <= 0:
 raise ValueError("Base e altura devem ser positivas")
self.base = base
 self.altura = altura

 @property
 def area(self) -> float:
 return self.base * self.altura

class Circulo:
 def _init_(self, raio: float):
 if raio <= 0:
 raise ValueError("Raio deve ser positivo")
self.raio = raio

 @property
 def area(self) -> float:
 return math.pi * (self.raio ** 2)

class FabricaFormas:
 @staticmethod
 def criar_forma(tipo: str, *args) -> Forma:
 formas = {
 'quadrado': Quadrado,
```

```
'retangulo': Retangulo,
'circulo': Circulo
}

if tipo not in formas:
raise ValueError(f"Tipo de forma não suportado: {tipo}")

return formas[tipo](*args)

# Exemplo de uso
if _name_ == "_main_":
try:
formas = [
FabricaFormas.criar_forma('quadrado', 4),
FabricaFormas.criar_forma('retangulo', 4, 5),
FabricaFormas.criar_forma('circulo', 3)
]

for forma in formas:
print(f"Área: {forma.area:.2f}")

except ValueError as e:
print(f"Erro: {e}")
```

Justificativas das Mudanças:

* Padrão Strategy: Cada forma agora é uma classe separada, seguindo o Open/Closed Principle. * Nomes descritivos: Variáveis e métodos têm nomes significativos (lado, base, altura, raio). * Tipagem estática: Adoção de type hints para melhor documentação e verificação. * Tratamento de erros: Validação de entradas inválidas.

* Factory Method: Padrão de criação para instanciar formas de maneira flexível. * Protocol/Interface: Definição clara do contrato que todas as formas devem seguir. * Princípio da Responsabilidade Única: Cada classe tem uma única responsabilidade.

3. Testes Unitários Implementados

```
python
import pytest
from formas import Quadrado, Retangulo, Circulo, FabricaFormas

class TestFormasGeometricas:
def test_area_quadrado(self):
quadrado = Quadrado(5)
assert quadrado.area == 25

def test_area_retangulo(self):
```

```
retangulo = Retangulo(4, 6)
assert retangulo.area == 24

def test_area_circulo(self):
circulo = Circulo(3)
assert round(circulo.area, 2) == 28.27

def test_lado_negativo_quadrado(self):
with pytest.raises(ValueError):
Quadrado(-5)

def test_fabrica_forma_invalida(self):
with pytest.raises(ValueError):
FabricaFormas.criar_forma('triangulo', 3, 4)
def test_fabrica_forma_valida(self):
quadrado = FabricaFormas.criar_forma('quadrado', 4)
assert isinstance(quadrado, Quadrado)
assert quadrado.area == 16
```

4. Conclusão

O Clean Code é importante para manutenção de software, porque promove a escrita de códigos claros, eficientes e organizados. O código seguindo essas normas é mais legível, não só pra quem mesmo escreveu, mas também para outros integrantes da equipe que possam usar ou modificar o código futuramente.

O Clean Code também contribui diretamente para a qualidade do software. Um texto mais legível é mais fácil de testar, refatorar e modificar, diminuindo a ocorrência de erros. Ao longo prazo, reduzindo custos em manutenção e proporcionando um desenvolvimento mais ágil.