Pensando com Antecedência: Introdução ao desenvolvimento orientado por testes (TDD)

TDD – Objetivos da aula

- Nesta aula vamos entender como o TDD guia o raciocínio lógico antes da implementação real do código.
- Ao final da aula você será capaz de:
  - Entender o conceito e o ciclo do **TDD**
  - Criar testes com **assert** para definir comportamentos
  - Relacionar comportamento esperado e implementação
  - Transformar requisitos em testes
  - Aplicar o TDD em um sistema de recompensas

Projeto AraCoins

- O AraCoins é um sistema de recompensas desenvolvido em Python, criado para apoiar a prática de Desenvolvimento Orientado por Testes (TDD).
- O sistema registra compras feitas por um usuário e converte valores em pontos baseados em regras como:
  - Percentual fixo por compra
  - Bônus para compras acima de certos valores
  - Acúmulo de pontos por usuário
  - Um cenário perfeito para aplicar TDD.

Por que usar TDD no projeto AraCoins?

- Ajuda a esclarecer regras antes de implementar
- Torna o comportamento explícito e verificável
- Reduz erros e retrabalho
- Mantém a lógica simples e incremental
- Facilita evolução futura do sistema

Origem e História do TDD

- Década de 1990: Ideias iniciais surgem na comunidade Smalltalk, com experimentos de “testar antes”.
- Kent Beck: Principal responsável por formalizar e popularizar o TDD dentro do Extreme Programming (XP).
- SUnit e JUnit: Kent Beck criou o SUnit (Smalltalk) e depois ajudou a inspirar o JUnit, que impulsionou testes automatizados.
- 2002: Publicação do livro “**Test-Driven Development: By Example**”, que consolidou o TDD como prática global de desenvolvimento ágil.

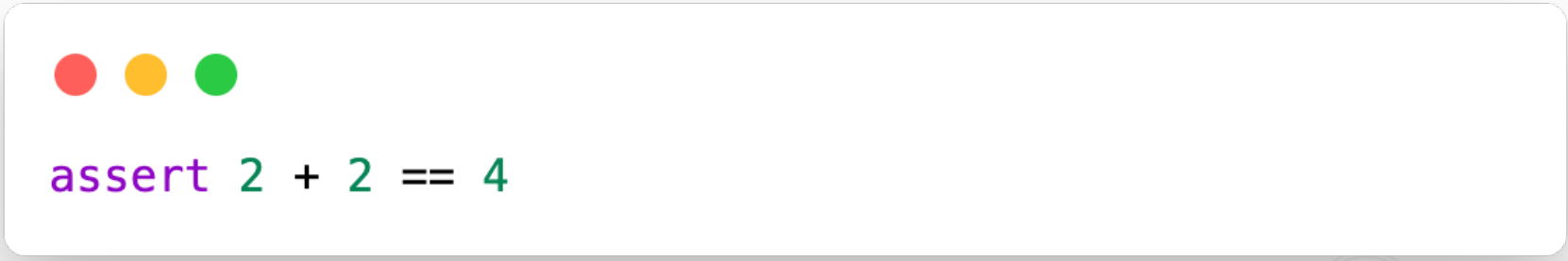
O Ciclo TDD: A Base da Metodologia

- O ciclo TDD é composto por três passos:
  - RED: escrever um teste que falha → valida o comportamento esperado
  - GREEN: implementar o mínimo necessário para fazê-lo passar
  - REFACTOR: melhorar o código preservando o comportamento

- Repetido continuamente até completar a funcionalidade.

Introdução ao assert

- O comando **assert** verifica se uma condição é verdadeira.
- Se for falsa, o teste falha.
- É a forma mais simples de expressar comportamento esperado.
- == é o operador de comparação de igualdade.



Usos comuns do assert

- Podemos testar:
  - valores
  - tipos
  - resultados de funções
  - conteúdo de listas, strings e dicionários

```
assert "a" in "abc"
assert len([1, 2, 3]) == 3
```

assert com Mensagens Explicativas

- Para depuração mais clara:

```
assert soma(2, 2) == 4, "A função soma deveria retornar 4"
```

```
assert "total" in calcula_compra({}), "O dicionário retornado deveria conter a chave 'total'"
```

```
assert formata_nome("joao") == "Joao", "A função formata_nome deveria capitalizar o nome"
```

Verificação de Tipos e Estruturas

- Exemplo 1 — Retorno deve ser inteiro

```
resultado = total_itens([1, 2, 3])
assert isinstance(resultado, int), "O total de itens deveria ser um inteiro"
```

- Exemplo 2 — Retorno deve ser um dicionário

```
dados = gerar_relatorio()
assert isinstance(dados, dict), "A função gerar_relatorio deveria retornar um dicionário"
```

Pensando com Antecedência

- Planejamento:
  - Antes de escrever código, pergunte:
  - O que esta função deve fazer?
  - Quais entradas são válidas?
  - Quais casos extremos podem ocorrer?
  - Como saberei se está funcionando?

- TDD responde isso criando testes antes da implementação.

Primeira Regra do Sistema AraCoins

- Toda compra gera 10% do valor em pontos.
  - Exemplos:
    - R$ 50 → 5 pontos
    - R$ 100 → 10 pontos
    - R$ 300 → 30 pontos

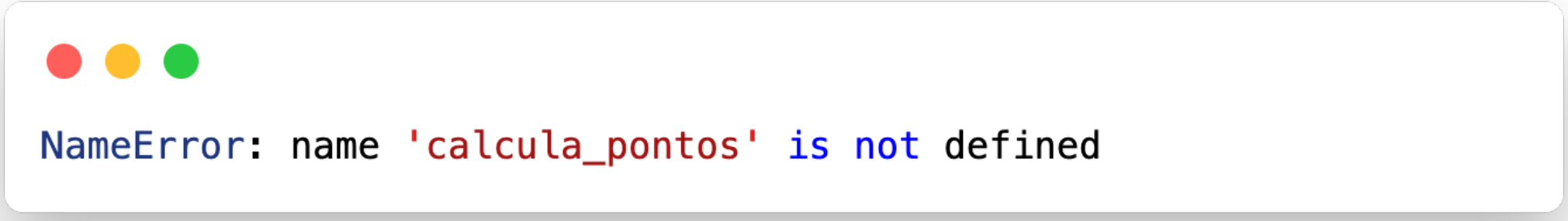
Primeiro Teste (Fase RED)

- Este teste define o comportamento inicial esperado da função: para uma compra de 100 unidades monetárias, o sistema deve calcular exatamente 10 pontos.
- Antes de escrever qualquer código, deixamos explícito o resultado esperado.

```python
def test_pontos_basico():
    assert calcula_pontos(100) == 10
```

Teste deve falhar (Fase RED)

- Como ainda não implementamos a função calcula_pontos, o teste falha imediatamente com um **NameError**. Essa falha é esperada na fase RED do TDD, pois confirma que o teste está realmente verificando algo que ainda não existe.



Implementação Mínima (GREEN)

- Implementamos apenas o suficiente para o teste passar: a função retorna 10% do valor informado.
- No TDD, essa implementação mínima é intencional, fazemos apenas o necessário para sair do estado RED.

```python
def calcula_pontos(valor):
    return valor * 0.10
```

Implementação Mínima (GREEN)

- Vamos inserir um novo requisito no sistema AraCoins:
  - Regra de bônus
    - Compras acima de R$ 500 recebem +50 pontos.
    - Como criar este teste antes da implementação?

Teste para o Bônus (RED)

- Criaremos um novo teste que descreve uma regra adicional: compras acima de 500 devem receber 50 pontos extras.
- Como essa lógica ainda não existe, o teste naturalmente começará falhando, exatamente como esperado na fase RED do TDD.

```python
def test_pontos_bonus():
    assert calcula_pontos(600) == 60 + 50
```

Ajuste da função (GREEN)

- Ajustamos a função para incluir a nova regra de negócio: se o valor da compra for maior que 500, adicionamos 50 pontos ao cálculo.
- Essa pequena mudança é suficiente para fazer o novo teste passar, completando a fase GREEN.

```python
def calcula_pontos(valor):
    pontos = valor * 0.10
    if valor > 500:
        pontos += 50
    return pontos
```

Testes como Documentação viva

- Os testes agora documentam:
  - regra dos 10%
  - regra de bônus
  - comportamento geral da função

- Os testes funcionam como uma documentação viva do sistema, pois descrevem de forma precisa e sempre atualizada como a função deve se comportar.

Testando múltiplos valores

- Ao testar vários valores diferentes, garantimos que a lógica da função funciona de forma consistente em cenários simples, médios e extremos, fortalecendo a confiança no comportamento geral do código.

```
assert calcula_pontos(50) == 5
assert calcula_pontos(200) == 20
assert calcula_pontos(1000) == 100 + 50
```

Funções auxiliares

- Criamos uma função auxiliar para isolar a regra de bônus e tornar o código mais legível. Testar essa função separadamente ajuda a garantir que cada parte da lógica do sistema esteja correta e fácil de manter.

```python
def eh_bonus(valor):
    return valor > 500

assert eh_bonus(600) is True
assert eh_bonus(120) is False
```

Testes de Erros e Exceções

- Também podemos testar situações inválidas. Neste exemplo, verificamos se a função lança um **ValueError** quando recebe um valor negativo. Isso garante que o sistema trate erros corretamente e evita comportamentos inesperados.

```
try:
    calcula_pontos(-10)
    assert False
except ValueError:
    assert True
```

```
# Na implementação
if valor < 0:
    raise ValueError("Valor inválido")
```

Pequenas iterações = Menos erros

- Cada teste adiciona uma regra.
- Cada regra é implementada isoladamente.
- O sistema cresce de forma segura.

- Esse ciclo controlado evita surpresas, reduz bugs e garante que cada avanço no código esteja sempre validado por testes automatizados.

Testando acúmulo de pontos

- Agora começamos a testar o comportamento do sistema em um nível mais alto, verificando se o acúmulo de pontos por usuário funciona corretamente quando várias compras são registradas.

```
usuario = Usuario()
usuario.add_compra(100)
usuario.add_compra(200)

assert usuario.total_pontos() == 10 + 20
```

Implementação mínima da Classe

- Criamos apenas a estrutura mínima da classe para que o teste possa rodar, seguindo o princípio do TDD de implementar o menor código possível antes do próximo ciclo de testes.

```python
class Usuario:
    def __init__(self):
        self.pontos = 0

    def add_compra(self, valor):
        self.pontos += calcula_pontos(valor)

    def total_pontos(self):
        return self.pontos
```

Testes de Estrutura

- Usamos **hasattr** para verificar se o objeto **Usuario** possui o atributo pontos. Junto com **callable**, garantimos que os métodos *add_compra* e *total_pontos* existem e podem ser chamados.
- Esses testes confirmam que a estrutura da classe está correta antes de focarmos na lógica interna.

```
u = Usuario()
assert hasattr(u, "pontos")
assert callable(u.add_compra)
assert callable(u.total_pontos)
```

Refatoração (REFACTOR)

- Agora é seguro melhorar o código:
  - Extrair funções
  - Remover duplicações
  - Criar constantes
  - Melhorar legibilidade

- Testes garantem que nada **quebra**.

Palavra-chave pass

- **pass**
  - Usado para definir funções ou classes temporárias sem implementar nada ainda.
  - Muito útil no TDD quando você quer criar o teste antes de implementar a função.

```python
def test_nivel_usuario():
    pass  # Ainda vamos escrever o teste
```

Testes Como Especificação Viva

- Testes mostram:
  - o que o sistema deve fazer
  - como ele deve reagir
  - quais cenários são esperados

- Os testes estão sempre atualizados automaticamente, ou seja, sempre que o código muda, os testes indicam imediatamente se algo deixou de funcionar, servindo como documentação viva e confiável.

Rodando os testes

- Agora vamos aprender a rodar os testes que criamos, verificando se nossas funções do **AraCoins** estão funcionando corretamente.

- Para isso, adicione todos os testes em um arquivo chamado **test_recompensas.py**. Em seguida, podemos executar todos os testes de uma vez ou testar casos específicos, garantindo que cada funcionalidade esteja funcionando como esperado.

Rodando os testes

- Importante: O **pytest** procura por funções e arquivos que seguem convenções de nomenclatura específicas:
  - **Arquivos**: o nome do arquivo precisa começar ou terminar com test, por exemplo:
    - test_recompensas.py ✅
    - recompensas_test.py ✅
    - recompensas.py ❌ (pytest não vai reconhecer automaticamente)
  - **Funções**: o nome precisa começar com test_, por exemplo:
    - def test_recompensa(): ✅
    - def teste_recompensa(): ❌

Rodando os testes

- Passo 1: Instalar o **pytest** com o comando:
  - pip install **pytest** (execute este comando no terminal do VS Code)
- Passo 2: Estrutura do arquivo de teste
  - Nome do arquivo: **test_recompensas.py**
  - Este arquivo contém todos os nossos testes criados. (Como test_pontos_basico, por exemplo)
- Passo 3: Rodar todos os testes
  - pytest (execute este comando no terminal do VS Code)

Rodando os testes

- Passo 4: Rodar um teste específico
  - pytest -k **test_pontos_basico**
  - Útil para focar em um teste isolado enquanto desenvolvemos.

- **Dica**: Também é possível rodar testes direto pelo VS Code com a extensão Python.
- Botões de “play” aparecem ao lado do teste para execução rápida.

Encerramento

- Você aprendeu:
  - O que é o TDD e seu ciclo
  - História e origem do TDD
  - Uso de assert (assertion)
  - Implementação guiada por testes
  - Construção do AraCoins via TDD
  - Como rodar seus testes no VS Code

PUC Minas