



# DESENVOLVIMENTO ORIENTADO A TESTES


Curso: Técnico em Desenvolvimento de Sistemas

Professor: Osires



# DOT – DESENVOLVIMENTO ORIENTADO A TESTES

- Filosofia/fundamentos do TDD (Test Driven Development) / DOT
- Passos do DOT
- O Comando Assert
- Exemplos de DOT em python
- DOT: Refatoração
- Classes de equivalência
- Exemplos de Classes de equivalências
- Testando valores inválidos
- Análise de valor limítrofe
- Desenvolvimento de programas em Python usando DOT



# Filosofia/fundamentos do TDD (Test Driven Development)

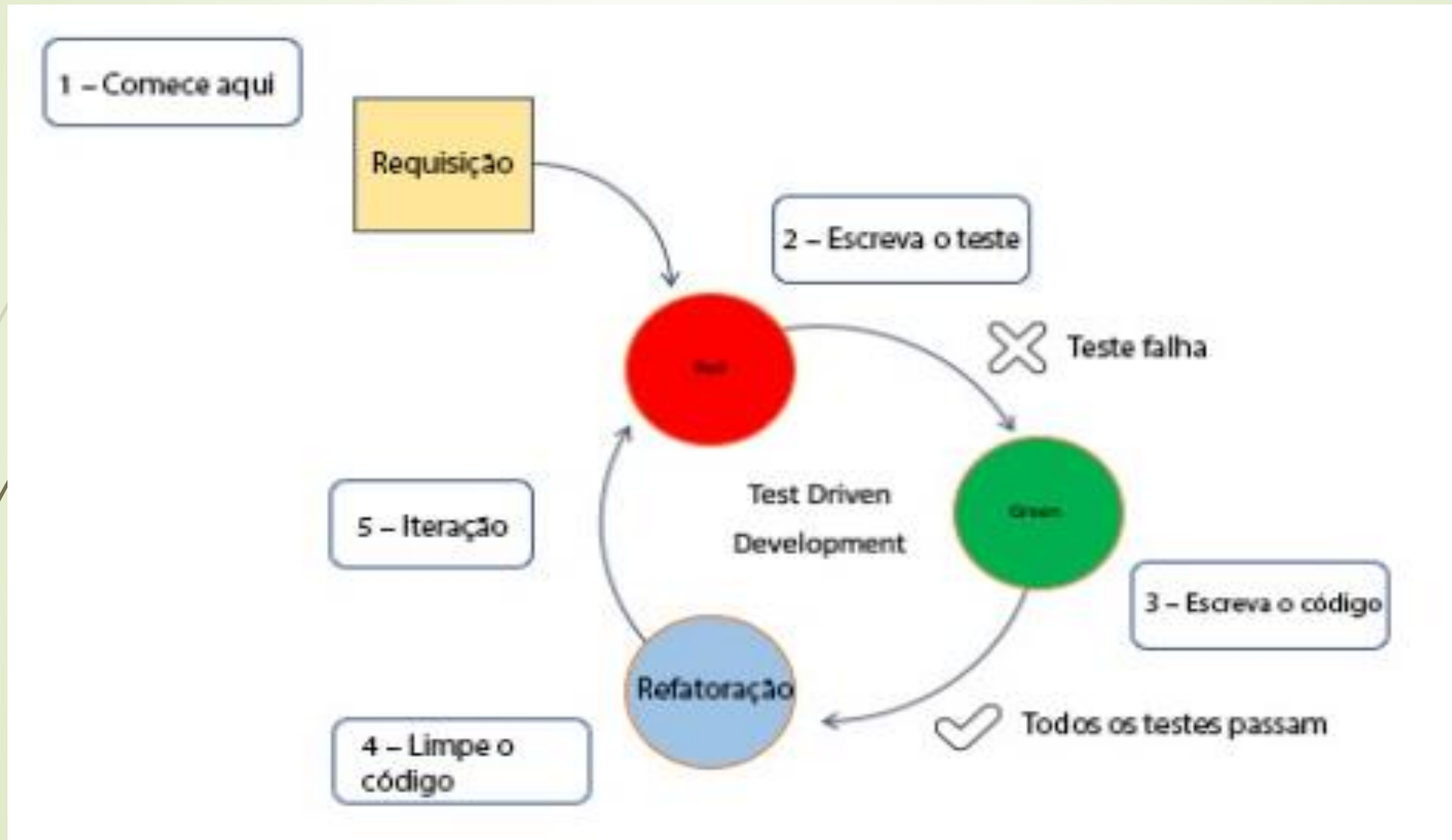
- Criada por Kent Back – EUA – 1999
- A ideia é pensar e criar os testes antes de programar
- Aplica-se os testes antes de desenvolver **uma nova função ou antes de modificá-la**
- Programador tem que estar muito “afinado” com as regras do negócio, saber que resultados a função deve retornar....



# Passos do DOT

- 1) Leia, entenda e analise o recurso a ser implementado ou o problema a ser resolvido.
- 2) Escrever os testes no qual o programa deve passar e depois executar o código de teste para vê-lo falhar pois a função ainda não existe
- 3) Escrever o código da função de forma que passe nos testes, corrigindo as falhas até que a função não encontre mais falhas
- 4) Após passar em todos os teste feitos, refatorar (melhorar) o código para deixa-lo mais eficiente
- 5) Voltar ao passo 1 e fazer tudo novamente.

# Passos do DOT





# O Comando Assert

- assert = afirmar
- Comando usado para testar uma condição
- Exemplo:

assert 2 == 2 (verdadeiro)

assert 3 == 2 (falso) programa é interrompido

- Em caso de Verdadeiro o programa continua normalmente
- Em caso de Falso o programa é interrompido e acusa uma exceção do tipo "AssertionError".

# Passo 1: escrever os testes usando o assert

## Valores para testes

Função: Fatorial	
Entrada x	Saida: fatorial(x)
5	120
3	6

```
def fatorial(x):  
    return
```

```
assert fatorial(3) == 6  
assert fatorial(5) == 120  
print("Todos os testes ok!")
```



# Passo 1: escrever os testes usando o assert

## Valores para testes

Função: Número primo	
Entrada x	Saida: Primo(x)
13	True
10	False

```
def primo(n):  
    return  
assert primo(13) == True  
assert primo(10) == False  
print("Todos os testes ok!")
```



## Passo 2: Executar o código de teste para vê-lo falhar pois a função ainda não existe

### Valores para testes

Função: Número primo	
Entrada x	Saida: Primo(x)
13	True
10	False

```
def primo(n):  
    return  
  
assert primo(13) == True  
print("Todos os testes ok!")
```

## Passo 3: Escrever o código da função de forma que **pass**e nos testes

```
def primo(n):  
    eh_primo = True  
    for i in range(2,n):  
        if (n % i == 0):  
            eh_primo = False  
    return eh_primo  
  
assert primo(13) == True  
assert primo(10) == False  
print("Todos os testes ok!")
```

## Passo 4: Executar os testes novamente, corrigindo as falhas até que a função não encontre mais falhas

```
def primo(n):  
    eh_primo = True  
    for i in range(2,n):  
        if (n % i == 0):  
            eh_primo = False  
    return eh_primo  
  
assert primo(13) == True  
assert primo(10) == False  
print("Todos os testes ok!")
```

## DOT: Refatoração

- Após a correção de todos os erros e o código passar em todos os testes propostos deve-se refatorar o código, se necessário.
- **Refatoração:** é o processo de modificar um sistema de software (código) para melhorar sua estrutura interna sem alterar seu comportamento externo.

Passo 5: Após passar em todos os teste feitos, refatorar (melhorar) o código para deixá-lo mais eficiente

## Função primo: código refatorado 1

```
def primo(n):  
    for i in range(2,n):  
        if (n % i == 0):  
            return False  
    return True  
  
assert primo(13) == True  
assert primo(10) == False  
assert primo(2) == True  
assert primo(1) == True  
print("Todos os testes ok!")
```



Passo 6: Repetir os testes até verificar que a refatoração não produziu novos erros.

### **Função primo: código refatorado 1**

```
def primo(n):  
    for i in range(2,n):  
        if (n % i == 0):  
            return False  
    return True  
  
assert primo(13) == True  
assert primo(10) == False  
assert primo(2) == True  
assert primo(1) == True  
print("Todos os testes ok!")
```



# Classes de equivalência

- Os casos de testes escolhidos são suficientes para saber que nenhum erro passou despercebido?
- Como saber?
- **Classes de equivalência:** Técnica de DOT que ajuda a diminuir a probabilidade de erros passarem despercebidos.
- A técnica consiste em identificar conjuntos de valores de testes considerados equivalentes entre si e diferenciar aqueles valores que podem ser considerados como casos especiais.
- Cada classe deve ser testada pelo menos uma vez.
- A quantidade de testes a serem escritos fica a critério da experiência do programador.



# Exemplo de Classes de equivalências

## Valores para testes

Função: Número primo	
Entrada x	Saida: Primo(x)
13	True
10	False

```
def primo(n):  
    return  
  
assert primo(13) == True # testando a classe de números primos  
assert primo(10) == False # testando a classe de números não primos  
print("Todos os testes ok!")
```

# Exemplo de Classes de equivalências

```
def primo(n):  
    for i in range(2,n):  
        if (n % i == 0):  
            return False  
    return True  
  
assert primo(13) == True # testando a classe de números primos  
assert primo(10) == False # testando a classe de números não primos  
print("Todos os testes ok!")
```



# Análise de valor limítrofe

- “Os bugs se escondem nas frestas”
- Análise do valor limítrofe é uma técnica complementar da técnica de Classes de Equivalência
- É utilizada quando os valores são numéricos ou ordenados
- Consiste em testar os valores limítrofes (limites) em cada classe de equivalência

# Análise de valores limítrofes: exemplo

Função que deve receber a idade de uma pessoa e retornar “menor” para idades menores que 18, “adulto” para idades entre 18 e 64 e “idoso” para idades maiores ou iguais a 65 anos.

```
def idade(x):  
    if x > 0 and x <= 17:  
        return "menor"  
    elif x >= 18 and x <= 64:  
        return "adulto"  
    else:  
        return "idoso"  
  
assert idade(1) == "menor" # testando classe válida, valor limítrofe  
assert idade(15) == "menor" # testando classe válida  
assert idade(17) == "menor" # testando classe válida, valor limítrofe  
assert idade(18) == "adulto" # testando classe válida, valor limítrofe  
assert idade(40) == "adulto" # testando classe válida  
assert idade(64) == "adulto" # testando classe válida, valor limítrofe  
assert idade(65) == "idoso" # testando classe válida, valor limítrofe  
assert idade(90) == "idoso" # testando classe válida  
print("Todos os testes ok!")
```



# Testando valores inválidos ou improváveis

- Valores inválidos são uma classe de equivalência de valores que podem ser passados para funções
- A função deve retornar o valor Exception quando receber valores inválidos ou improváveis.
- Ver exemplos em python

# Testando valores inválidos ou improváveis

"""Lista3\_q3. Faça uma função que recebe por parâmetro um valor inteiro e positivo e retorna o valor lógico Verdadeiro caso o valor seja primo e Falso em caso contrário."""

```
def primo(n):
```

```
    if type(n) != int or n <= 0:
```

```
        return Exception
```

```
    for i in range(2,n):
```

```
        if (n % i == 0):
```

```
            return False
```

```
    return True
```

```
assert primo("*") == Exception # testando a classe de valores inválidos
```

```
assert primo(3.5) == Exception # testando a classe de valores inválidos
```

```
assert primo(-1) == Exception # testando a classe de valores improváveis
```

```
assert primo(0) == Exception # testando a classe de valores improváveis
```

```
assert primo(1) == True # testando a classe de números primos
```

```
assert primo(2) == True # testando a classe de números primos
```

```
assert primo(13) == True # testando a classe de números primos
```

```
assert primo(10) == False # testando a classe de números não primos
```

```
print("Todos os testes ok!")
```