Estruturando o Problema com TDD: Classe e Modelo de Dados

Objetivos da aula

- Nesta aula vamos estruturar um projeto de Cinema em Dados usando TDD como guia do raciocínio lógico.
- Ao final da aula você será capaz de:
  - Entender o ciclo do TDD aplicado a um problema real
  - Definir comportamentos antes da implementação
  - Criar uma classe Filme orientada a dados reais
  - Usar estruturas nativas do Python de forma consciente

Problema

- O Problema: Analisar o Sucesso de Filmes
  - Queremos entender o que influencia o sucesso financeiro de um filme.
  - O projeto busca analisar dados reais de filmes para investigar padrões de sucesso financeiro.

O Que é Sucesso no Cinema?

- Cinema em dados:
  - Iremos definir sucesso financeiro como a diferença entre receita e orçamento.

$$sucesso = receita - orçamento$$

Pensando em TDD

- Primeiro escrevemos o teste, depois o código.
- No TDD (*Test-Driven Development*), o comportamento esperado é definido antes da implementação.

Primeiro Requisito do Sistema

- Um filme deve ter título, orçamento e receita.
- Antes de codar, identificamos claramente os requisitos da classe.

Criando o Primeiro Teste

- Este teste define o comportamento esperado ao criar um filme.

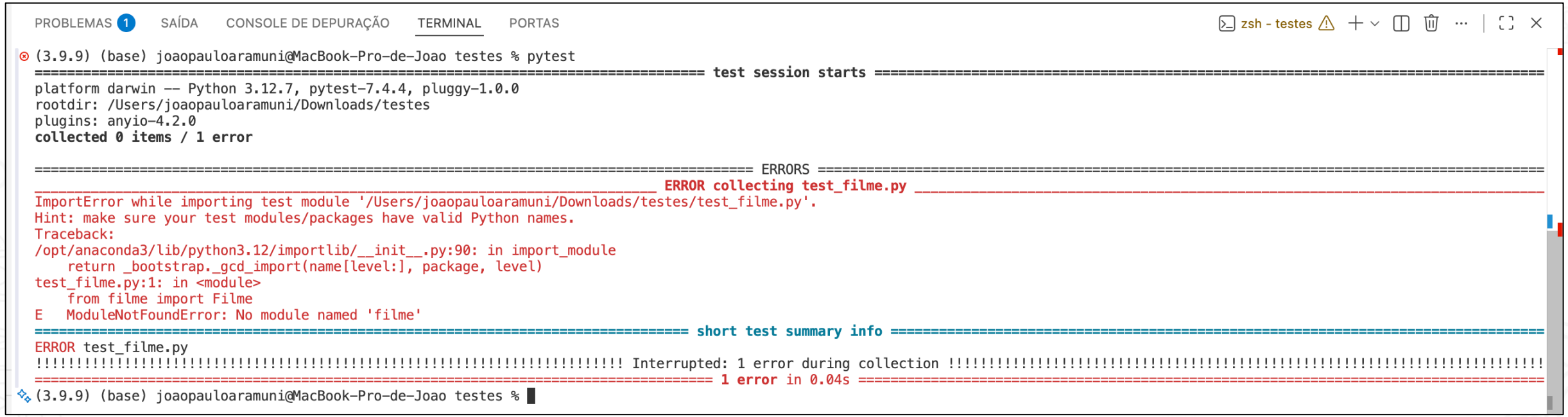
```python
from filme import Filme

def test_criacao_filme():
    filme = Filme("Matrix", 63000000, 465000000)
    assert filme.titulo == "Matrix"
```

Teste Falhando (Estado Vermelho)

- *NameError: Filme is not defined*
  - O teste falha porque a classe ainda não foi implementada.
- Para testar utilize:
  - *Rodar todos os testes*
    - *pytest test_filme.py ou simplesmente pytest*
  - *Rodar apenas um teste*
    - *pytest test_filme.py::test_criacao_filme*

Teste Falhando (Estado Vermelho)

- Terminal do VS Code:



Implementação Mínima da Classe Filme

- Criamos apenas o código necessário para passar no teste.

```
class Filme:
    def __init__(self, titulo, orcamento, receita):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
```

O método init (Construtor da Classe)

- O método **__init__** é executado automaticamente na criação do objeto e inicializa os atributos de cada filme.

```
class Filme:
    def __init__(self, titulo, orcamento, receita):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
```

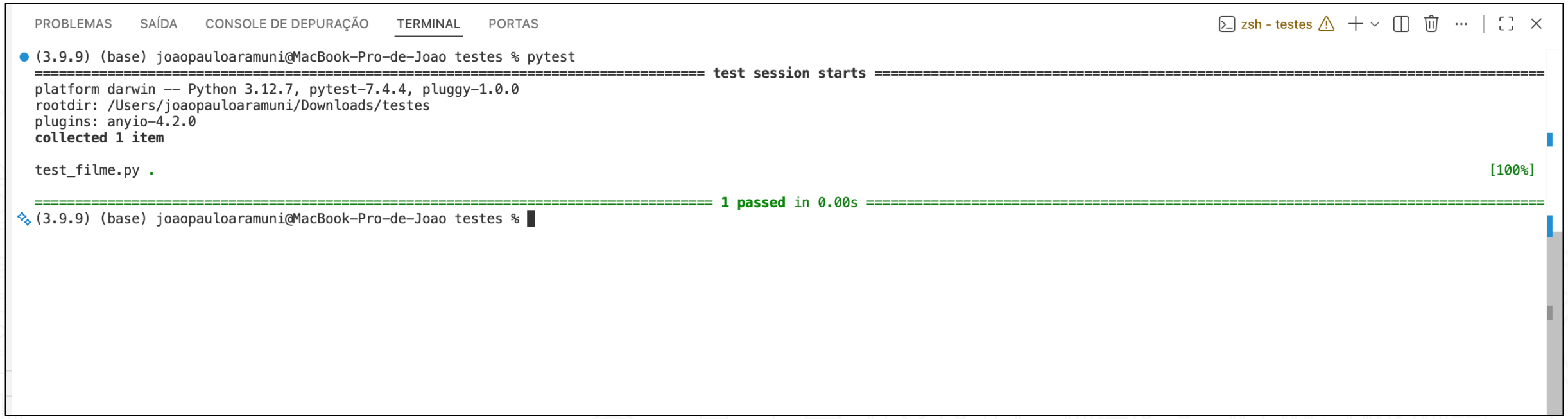
O que é o self no construtor?

- O **self** representa o próprio objeto criado e é usado para acessar e armazenar atributos que pertencem a cada filme.

```
class Filme:
    def __init__(self, titulo, orcamento, receita):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
```

Teste Passando (Estado Verde)

- Teste executado com sucesso.
- Quando o teste passa, validamos o comportamento implementado.



Refatoração com Segurança

- Nenhuma refatoração necessária por enquanto.
- No TDD, só refatoramos quando o teste está verde.

Adicionando Novo Atributo: Gênero

- Um novo requisito surge a partir da análise do domínio do problema.

```
filme = Filme("Titanic", 200000000, 2200000000, "Drama")
```

Teste para o Novo Atributo

- O teste documenta o novo comportamento esperado.

```python
def test_filme_possui_genero():
    filme = Filme("Titanic", 200000000, 2200000000, "Drama")
    assert filme.genero == "Drama"
```

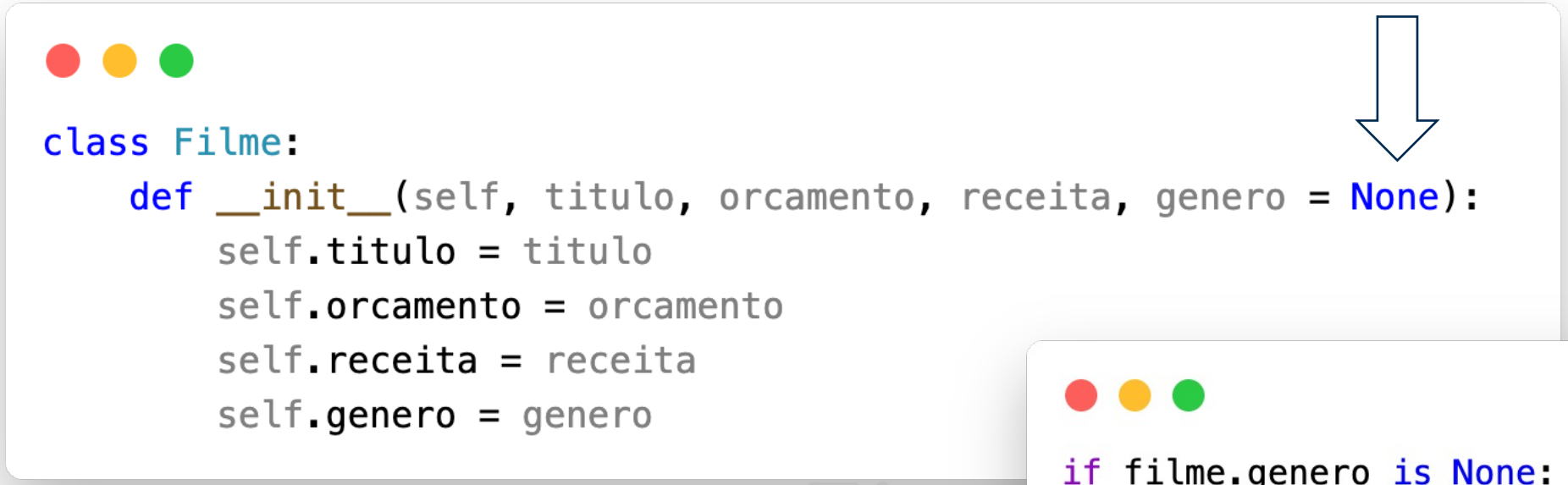
Atualizando o Construtor

- Ajustamos a classe para atender ao novo teste.

```
class Filme:
    def __init__(self, titulo, orcamento, receita, genero):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
        self.genero = genero
```

Uso de None para Gênero Desconhecido

- O valor **None** indica a ausência de informação, permitindo criar filmes mesmo quando o gênero ainda não é conhecido.



```python
class Filme:
    def __init__(self, titulo, orcamento, receita, genero = None):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
        self.genero = genero
```

```python
if filme.genero is None:
    print("Gênero não informado")
```

Testando Tipos Numéricos com TDD

- O teste valida o comportamento do método e garante que dados financeiros inválidos sejam detectados.

```python
def test_valores_sao_numericos():
    filme = Filme("Interestelar", 165000000, 677000000, "Ficção")
    assert filme.valores_sao_numericos() is True
```

Usando Tipos Numéricos

- O método encapsula a validação garantindo que orçamento e receita sejam valores numéricos.

```
def valores_sao_numericos(self):
    return isinstance(self.orcamento, (int, float)) and
    isinstance(self.receita, (int, float))
```

Teste para o Método lucro

- O teste valida o comportamento do método de cálculo do lucro.

```python
def test_calculo_lucro():
    filme = Filme("Joker", 55000000, 1074000000, "Drama")
    assert filme.lucro() > 0
```

Calculando o Lucro do Filme

- Criamos um método que encapsula a regra de negócio.

```
def lucro(self):
    return self.receita - self.orcamento
```

Encapsulamento de Comportamento

- O cálculo do lucro pertence à classe Filme.
- O comportamento fica junto dos dados que ele manipula.
  - Para conhecimento: Aqui estamos aplicando o padrão GRASP - Especialista.

Classe Filme até agora: filme.py

```python
class Filme:
    def __init__(self, titulo, orcamento, receita, genero):
        self.titulo = titulo
        self.orcamento = orcamento
        self.receita = receita
        self.genero = genero

    def valores_sao_numericos(self):
        return isinstance(self.orcamento, (int, float)) and
        isinstance(self.receita, (int, float))

    def lucro(self):
        return self.receita - self.orcamento
```

Testando str

- Testamos a representação textual do objeto. (VERMELHO)

```
def test_str_filme():
    filme = Filme("Avatar", 237000000, 2847000000, "Ficção")
    assert "Avatar" in str(filme)
```

Representação em Texto do Filme

- O método **__str__** facilita a visualização do objeto. (VERDE)

```
def __str__(self):
    return f"{self.titulo} ( {self.genero} )"
```

Script de testes até agora:

test_filme.py

```python
from filme import Filme

def test_criacao_filme():
    filme = Filme("Matrix", 63000000, 465000000, "Ficção")
    assert filme.titulo == "Matrix"

def test_filme_possui_genero():
    filme = Filme("Titanic", 200000000, 2200000000, "Drama")
    assert filme.genero == "Drama"

def test_valores_sao_numericos():
    filme = Filme("Interestelar", 165000000, 677000000, "Ficção")
    assert filme.valores_sao_numericos() is True

def test_calculo_lucro():
    filme = Filme("Joker", 55000000, 1074000000, "Drama")
    assert filme.lucro() > 0

def test_str_filme():
    filme = Filme("Avatar", 237000000, 2847000000, "Ficção")
    assert "Avatar" in str(filme)
```

Lista de Filmes

- filmes = []
  - Usaremos listas para armazenar coleções de filmes.

```
filmes =  [
    Filme("Matrix", 63000000, 465000000, "Ficção"),
    Filme("Titanic", 200000000, 2200000000, "Drama"),
    Filme("Avatar", 237000000, 2847000000, "Ficção")
]
```

Lista de Filmes

- Adicionando Filmes à Lista: filmes.**append**(filme).
  - A lista permite crescimento dinâmico dos dados.

```
filmes = []

filme = Filme("Matrix", 63000000, 465000000, "Ficção")
filmes.append(filme)

filmes.append(Filme("Titanic", 200000000, 2200000000, "Drama"))
```

Percorrendo a Lista

- Iteramos sobre a estrutura para acessar os objetos.

```
for filme in filmes:
    print(filme.titulo)
```

Separando Responsabilidades

- Cada módulo possui uma responsabilidade clara.
  - **filme.py** -> classe Filme
  - **test_filme.py** -> script de testes
  - **main.py** -> lógica do programa

Separando Responsabilidades: main.py

```python
from filme import Filme

def main():
    filmes = []

    filmes.append(Filme("Matrix", 63000000, 465000000, "Ficção"))
    filmes.append(Filme("Titanic", 200000000, 2200000000, "Drama"))
    filmes.append(Filme("Avatar", 237000000, 2847000000, "Ficção"))
    filmes.append(Filme("Coringa", 55000000, 1074000000))  # gênero não informado

    for filme in filmes:
        print(filme) # o Python chama automaticamente o método __str__ do objeto
        print(f"Lucro: {filme.lucro()}")
        print("-" * 30)

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

Encerramento

- Você aprendeu:
  - Como estruturar um problema real usando **TDD**
  - Criar uma classe Filme orientada a dados de cinema
  - Uso de **assert** (assertion)
  - Aplicar encapsulamento e modularização
  - Usar **listas** para armazenar objetos
  - Preparar a base para análises, visualizações e predições

PUC Minas