Teste antes do código

Aula 03 — TDD: vermelho, verde, refatora

Aula 03 · 06/11/2026 · Indicadores 1, 2

UniSenac Centro Universitário RS
Senac
Fecomércio
IFEP
Sesc
Sindicatos Empresariais

O mapa de hoje

- **Bloco 1** — testar do jeito de sempre, e depois ao contrário
- **Bloco 2** — o código mais simples que passa, e arrumar sem quebrar
- **Bloco 3** — passo pequeno, código melhor, e quando não vale a pena
- **Bloco 4** — mão na massa: TDD em dupla, na locadora e no projeto de vocês

> Baixem o `emprestimo.py` da página da aula e abram a pasta no VS Code. Hoje ele chega **sem teste nenhum**.

Retomando a Aula 02

- Um teste que passava ontem ficou vermelho hoje, e ninguém mexeu nele. O que esse vermelho está contando?
- A multa começa no **3º dia** de atraso. Testes com 2, 3 e 4 dias: qual dos três pega quem escreveu `>` no lugar de `>=`?
- Quando um teste deve **esperar** um erro, em vez de fugir dele? Como o pytest escreve isso?

Bloco 1

Testes, do jeito que vocês já sabem

O empréstimo do balcão, sem teste nenhum

```
class Emprestimo:
    def __init__(self, cliente):
        self.cliente = cliente
        self.jogos = []

    def adicionar(self, jogo):
        self.jogos.append(jogo)
```

É pequena. Vale a pena testar? O que acontece no balcão se o `adicionar` estiver errado?

O cliente entra quando o empréstimo é criado, e cada jogo entra depois — igual às tabelas `emprestimo` e `item_emprestimo` da UC2.

Testando depois, como sempre

```python
from emprestimo import Cliente, Emprestimo

def test_recebe_um_jogo():
    emprestimo = Emprestimo(Cliente("Ana"))
    emprestimo.adicionar("Zelda")
    assert emprestimo.jogos == ["Zelda"]

def test_recebe_varios_jogos():
    emprestimo = Emprestimo(Cliente("Ana"))
    emprestimo.adicionar("Zelda")
    emprestimo.adicionar("FIFA 25")
    assert emprestimo.jogos == ["Zelda", "FIFA 25"]
```

Criem `test_emprestimo.py` **na mesma pasta**. Antes do ▶: verde ou vermelho?

Chegaram duas regras do balcão

- O **mesmo jogo** não entra duas vezes no mesmo empréstimo
- Um empréstimo leva **no máximo 3 jogos**

Do jeito de sempre, vocês mexeriam no `adicionar` e testariam depois. Hoje vai ser ao contrário: **o teste vem primeiro**.

> O banco da UC2 já recusa o jogo repetido: a chave primária de `item_emprestimo` é `(id_emprestimo, id_jogo)`. Mas o atendente só descobriria isso na hora de gravar, com o cliente esperando no balcão.

O teste de uma regra que ainda não existe

```python
import pytest

def test_nao_aceita_o_mesmo_jogo_duas_vezes():
    emprestimo = Emprestimo(Cliente("Ana"))
    emprestimo.adicionar("Zelda")
    with pytest.raises(ValueError):
        emprestimo.adicionar("Zelda")
    assert emprestimo.jogos == ["Zelda"]
```

O `adicionar` continua igual. Antes de rodar, combinem na dupla: **verde ou vermelho?** E se for vermelho, por que isso seria uma boa notícia?

⚠ Vermelho — e era isso que a gente queria

```
FAILED test_nao_aceita_o_mesmo_jogo_duas_vezes
E       Failed: DID NOT RAISE ValueError
```

GLOSSÁRIO

`FAILED`	falhou — o teste ficou vermelho
`DID NOT RAISE`	não levantou — o erro que o teste esperava não aconteceu

O vermelho conta duas coisas:

- a regra ainda **não existe** no código
- o teste **é capaz de perceber** que ela não existe

Checagem do Bloco 1

Agora a segunda regra, ainda **sem** mexer no `emprestimo.py`:

- Escrevam o teste `test_nao_aceita_mais_de_3_jogos`
- Antes de rodar: verde ou vermelho? Em que linha o pytest vai apontar?
- Rodem. O palpite bateu? Se não bateu, o que vocês esperavam diferente?



`forms.gle/rT66vgb8aNEvGTjx8`

Intervalo

10 minutos

Bloco 2

Do vermelho ao verde, e arrumar sem quebrar

O código mais simples que deixa verde

```
    def adicionar(self, jogo):
        if jogo in self.jogos:
            raise ValueError("Esse jogo já está no empréstimo.")
        self.jogos.append(jogo)
```

Só isso. Nada de limite de 3 ainda: aquele teste continua vermelho, e está certo que continue.

Rodem os **quatro** testes. Quantos ficam verdes? E os dois de antes do intervalo, continuam verdes?

O limite de 3, do mesmo jeito

```
    def adicionar(self, jogo):
        if jogo in self.jogos:
            raise ValueError("Esse jogo já está no empréstimo.")
        if len(self.jogos) >= 3:
            raise ValueError("Já são 3 jogos.")
        self.jogos.append(jogo)
```

Palpite antes do ▶: quantos verdes agora?

O `>=` não é distração. Com 3 jogos na lista, o quarto já não entra. Com `>`, entraria — e só o teste do limite perceberia.

Está verde. Dá para ler melhor?

```
    def adicionar(self, jogo):
        if self._ja_tem(jogo):
            raise ValueError("Esse jogo já está no empréstimo.")
        if self._lotado():
            raise ValueError(f"Já são {LIMITE_DE_JOGOS} jogos.")
        self.jogos.append(jogo)

    def _ja_tem(self, jogo):
        return jogo in self.jogos

    def _lotado(self):
        return len(self.jogos) >= LIMITE_DE_JOGOS
```

O 3 virou a constante `LIMITE_DE_JOGOS`. Como vocês **sabem** que nada quebrou?

O que vocês acabaram de fazer tem nome

1. **Escrever o teste** de uma regra que ainda não existe
2. Rodar e **ver ficar vermelho**
3. Escrever o **código mais simples** que passa
4. Rodar e **ver ficar verde** — o teste novo e todos os antigos
5. **Refatorar**, e rodar de novo

GLOSSÁRIO

TDD — ***Test-Driven Development***	desenvolvimento guiado por testes
red · green · refactor	vermelho · verde · refatorar — as três fases do ciclo
refactor	refatorar — melhorar o código sem mudar o que ele faz

Este teste está certo?

```python
def test_nao_aceita_o_mesmo_jogo_duas_vezes():
    emprestimo = Emprestimo(Cliente("Ana"))
    emprestimo.adicionar("Zelda")
    emprestimo.adicionar("Zelda")
    assert "Zelda" in emprestimo.jogos
```

Ele foi escrito **antes** da regra, contra o `emprestimo.py` que vocês baixaram hoje cedo.

Palpite na dupla: verde ou vermelho? E depois que a regra existir?

⚠ Erro comum — o teste que nunca ficou vermelho

> Ele fica **verde sem a regra**: `"Zelda" in jogos` é verdade com um Zelda e com dois. E fica **vermelho depois que a regra existe**, porque o segundo `adicionar` agora levanta o erro. O teste está de cabeça para baixo, e só quem o viu verde na hora errada percebe.

Por isso o passo 2 do ciclo não é burocracia: **ver o teste falhar é testar o teste**. Teste também é código, e código pode estar errado.

- Ficou verde de primeira, sem código novo? Desconfiem
- Quebrem o código de propósito e vejam se o teste percebe

Checagem do Bloco 2

- A dupla escreveu um teste novo e ele ficou **verde na primeira rodada**, sem código novo. O que vocês fazem agora?
- Na refatoração, um teste antigo ficou vermelho. Vocês consertam o **teste** ou o **código**? Por quê?
- Escrevam o código mais simples que deixa este teste verde:

```
def test_nao_aceita_jogo_sem_titulo():
    emprestimo = Emprestimo(Cliente("Ana"))
    with pytest.raises(ValueError):
        emprestimo.adicionar("")
```

forms.gle/i166rwZTDejWwR8z5

Intervalo

10 minutos

Bloco 3

Passo pequeno, código melhor

Regra nova: o cliente fiel leva 4

Quem já fez **mais de 10 empréstimos** leva até 4 jogos. A lista de testes já está aqui:

- O cliente fiel não leva o 5º jogo
- Um cliente com 11 empréstimos leva 4 jogos
- O cliente novo continua levando 3
- Um cliente com exatamente 10 empréstimos ainda leva 3
- O cliente fiel continua barrado no jogo repetido

Na dupla: **qual vocês escrevem primeiro?** E qual deixam para o fim?

⚠ Erro comum — começar pelo caso mais difícil

> O reflexo é começar pelo "fiel não leva o 5º": é o mais interessante, e é o que obriga a escrever mais código de uma vez. Se ele ficar vermelho por três motivos ao mesmo tempo, qual deles vocês consertam primeiro?

Comecem pelo mais simples — "o cliente novo continua levando 3" — e subam **um degrau por ciclo**. Passo maior só quando a confiança no código está alta, e isso é exceção.

GLOSSÁRIO

baby steps	passos de bebê — o menor teste e o menor código que fazem o ciclo andar

O teste é o primeiro a usar o seu código

Dois jeitos de testar a fronteira dos 10 empréstimos:

```python
def test_cliente_com_10_ainda_leva_3():
    emprestimo = Emprestimo(Cliente("Caio", 10))
    for jogo in ["Zelda", "FIFA 25", "Hades"]:
        emprestimo.adicionar(jogo)
    with pytest.raises(ValueError):
        emprestimo.adicionar("Celeste")
```

```python
def test_cliente_com_10_ainda_leva_3():
    assert Cliente("Caio", 10).limite_de_jogos() == 3
```

Qual vocês prefeririam escrever dez vezes? O segundo só existe se o limite **morar no cliente**. O teste difícil de escrever estava avisando onde a regra devia ficar.

E o `_ja_tem`, eu testo?

```python
def test_ja_tem():
    emprestimo = Emprestimo(Cliente("Ana"))
    emprestimo.adicionar("Zelda")
    assert emprestimo._ja_tem("Zelda")
```

O `_` na frente, em Python, quer dizer **de uso interno**. Quem está de fora — o balcão, a tela, o teste — usa o `adicionar`.

> Não testem o privado. O teste do `adicionar` já passa por ele. Se der vontade de testar o privado sozinho, é o código dizendo que aquilo está no lugar errado: tirem para onde ele pertence, como o limite que foi morar no `Cliente`.

⚠ Erro comum — "esse é simples demais para testar"

```
    def _lotado(self):
        return len(self.jogos) >= LIMITE_DE_JOGOS
```

Uma linha. Troquem o `>=` por `>`: o que muda no balcão amanhã?

> Todo código parece simples na semana em que foi escrito: ele está inteiro na cabeça de quem escreveu. Um mês depois, não está mais. Se tem regra de negócio, tem teste — e a fronteira da Aula 02 mora justamente nessas linhas "simples".

TDD o tempo todo?

VALE A PENA

- Regra nova, como as de hoje
- Corrigir um defeito: primeiro o teste que mostra o defeito, depois o conserto
- Código complicado, que dá medo de mexer

PODE NÃO VALER

- Escrever os testes que faltaram num código que já funciona — isso é escrever teste, não é TDD
- Código trivial de verdade, sem regra nenhuma

> Cuidado com o "aqui é tudo trivial". Quem acha isso de todo código nunca faz TDD.

Checagem do Bloco 3

Regra da confeitaria: **encomenda precisa de pelo menos 2 dias de antecedência**.

- Ordenem do teste mais simples ao mais difícil: encomenda para daqui a 10 dias · para hoje · para daqui a exatamente 2 dias · para ontem
- TDD ou não? (a) corrigir a multa que está cobrando no dia do prazo · (b) escrever os testes que faltaram no CRUD da UC2
- Um colega escreveu `assert encomenda._prazo_ok()`. O que vocês diriam a ele?



forms.gle/WHHiFX3daZeZJtLbA

Intervalo

10 minutos

MÃO NA MASSA

TDD em dupla

Uma pessoa escreve o **teste**, a outra escreve o **código** que deixa verde. A cada ciclo, trocam de lugar.

Cada ciclo vira uma linha no topo do arquivo de teste:

```
# ciclo | teste escrito                | falhou? | código escrito              | passou?
# 1     | test_cliente_novo_leva_3     | sim     | limite_de_jogos() dá 3      | sim
# 2     | ...                          |         |                             |
```

GLOSSÁRIO

pair programming	programação em dupla — duas pessoas, um teclado, papéis trocados

MÃO NA MASSA

Parte A — o cliente fiel, na locadora

Quem já fez **mais de 10 empréstimos** leva até **4 jogos**.

1. Peguem a lista do Bloco 3 e comecem pelo teste **mais simples**
2. Um ciclo por teste — vermelho, verde, refatora — e trocam de lugar
3. Não esqueçam a **fronteira**: com 10 empréstimos, ainda leva 3
4. Anotem cada ciclo na tabela, inclusive o vermelho
5. No fim, **todos** os testes verdes, os de hoje cedo também

> Viram verde de primeira? Parem e descubram por quê antes de seguir.

MÃO NA MASSA

Parte B — uma regra do projeto de vocês

Escolham uma regra do minimundo do grupo, que o CRUD da UC2 deveria respeitar. Por exemplo:

- **Futsal:** um time inscreve no máximo 12 jogadores
- **Confeitaria:** encomenda com pelo menos 2 dias de antecedência
- **RPG:** um personagem não passa do nível 20

1. Escrevam a regra numa frase, em português, no topo do `regra.py`
2. Listem os testes do mais simples ao mais difícil, no `test_regra.py`
3. TDD em dupla, como na Parte A, com a tabela

> Python puro, **sem banco**: a regra recebe os valores e devolve a resposta. Ligar no PostgreSQL fica para a Aula 05.

MÃO NA MASSA

O combinado desta entrega

GLOSSÁRIO

`grupoNN-aulaNN-artefato.<ext>`	`grupo03-aula03-tdd.zip`

A pasta da aula, compactada, com os quatro arquivos: `emprestimo.py`, `test_emprestimo.py`, `regra.py` e `test_regra.py`. Não juntem tudo num `.py` só: com esse nome, o painel Testing não encontra os testes.

O que eu confiro: a tabela tem **um vermelho antes de cada verde** · cada vermelho falhou pelo motivo certo · o teste da fronteira dos 10 está lá · tudo verde no fim · a regra do grupo cabe numa frase em português

> Terminou? Tentem testar o cliente fiel com o número de empréstimos vindo do banco, e não digitado. Vão travar, e é de propósito: é a Aula 04.

MÃO NA MASSA

Devolutiva — passo de mesa em mesa

Eu passo em cada dupla. Deixem na tela:

- A tabela do ciclo, com o primeiro vermelho
- O teste da fronteira, e por que aquele número
- A regra do grupo, em português e em teste

> Vale trocar os papéis também aqui: quem escreveu o código conta o que o teste cobra, e quem escreveu o teste conta o que o código faz.

PARA CASA · OPCIONAL

Antes da próxima aula

Testar em casa: escolham outra regra do projeto e façam dois ciclos sozinhos, com a tabela. Comecem pelo teste mais simples.

Tarefa opcional: abram o CRUD da UC2 e procurem uma função que vocês **não conseguiriam** testar sem o banco ligado. Anotem por quê.

Para ler: ANICHE, *Testes automatizados de software*, capítulo 2 — o mesmo ciclo, num sistema de leilão.

Para ver acontecer: demonstrações da Aula 03 — ver o vermelho chegar antes do código, e quebrar o código de propósito para ver quem percebe — e oito perguntas com resposta na hora.

Para a próxima aula

Hoje o número de empréstimos do cliente entrou digitado. **Como testar a regra que precisa do banco e do WhatsApp?**

UniSenac Centro Universitário RS
Senac
Fecomércio
IFEP
Sesc
Sindicatos Empresariais