

Departamento de Engenharia de Sistemas e Computação

Refatoração e Testes

Legibilidade, organização, robustez e boas práticas

João Araujo

Algoritmos Computacionais I

Aula 33

Objetivos da aula

- **Refatorar**: melhorar a legibilidade, a organização e a eficiência *sem mudar o comportamento*
- **Testar** casos extremos para garantir a robustez
- Aplicar boas práticas: nomes, coesão, modularização e separação de responsabilidades
- Receber feedback detalhado sobre as melhorias e ajustes finais

Cap. 9 em ação

Preocupe-se com seu leitor; seja simples; seja desapegado; teste seu programa.

O que é refatorar?

- Reorganizar o código **sem alterar o que ele faz**
- Como garantir isso? Com testes: rode-os antes e depois
- Faça mudanças *pequenas*, uma por vez

Fluxo seguro

1. Garanta que os testes passam
2. Faça *uma* melhoria
3. Rode os testes de novo
4. Repita

Sinais de que é hora de refatorar

- Funções com $>$ 30 linhas ou que fazem várias coisas
- Código **repetido** (copiar e colar)
- Nomes como `x`, `aux`, `dados2`
- Muitos níveis de `if`/`for` aninhados
- Números "mágicos" soltos no código
- Variáveis globais modificadas em várias funções
- Comentários explicando código confuso
- Funções que leem do teclado *e* calculam

Antes e depois: nomes e números mágicos

Antes

```
def f(d):
    t = 0
    for k in d:
        t = t + d[k][1] * d[k][2]
    if t > 5000:
        t = t * 0.95
    return t
```

Depois

```
LIMITE_DESCONTO = 5000.0
FATOR_DESCONTO = 0.95

def custo_total(estoque):
    """Custo do estoque, com desconto
    para compras acima do limite."""
    total = sum(item["preco"] * item["qtd"]
                for item in estoque.values())
    if total > LIMITE_DESCONTO:
        total *= FATOR_DESCONTO
    return total
```

Nomes que explicam, constantes nomeadas (Aula 5), docstring, compreensão no lugar de laço manual.

Antes e depois: eliminar repetição

Antes

```
p = float(input("Preço: "))
while p < 0:
    p = float(input("Preço: "))
q = float(input("Quantidade: "))
while q < 0:
    q = float(input("Quantidade: "))
m = float(input("Mínimo: "))
while m < 0:
    m = float(input("Mínimo: "))
```

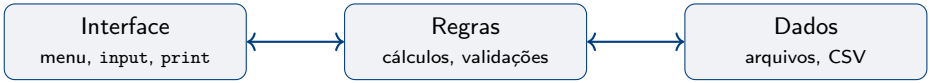
Depois

```
def le_nao_negativo(msg):
    while True:
        try:
            v = float(input(msg))
            if v >= 0:
                return v
        except ValueError:
            pass
        print("Valor inválido.")

preco = le_nao_negativo("Preço: ")
qtd = le_nao_negativo("Quantidade: ")
minimo = le_nao_negativo("Mínimo: ")
```

DRY (*don't repeat yourself*): se você copiou e colou, provavelmente falta uma função.

Separação de responsabilidades



- **Regras** não deveriam usar `input/print`: recebem parâmetros e devolvem valores
- **Interface** não deveria calcular: chama as regras
- **Dados** sabem ler e gravar; não sabem o que significam os números
- Resultado: cada parte é testável e substituível

Testes automáticos com `assert`

```python
# testes.py
from estoque import custo_total, baixa_estoque, insumos_em_alerta

def novo():
    return {"cimento": {"unidade": "saco", "preco": 10.0, "qtd": 5, "minimo": 3}}

def teste_custo_total():
    assert custo_total(novo()) == 50.0
    assert custo_total({}) == 0.0                # caso-limite: vazio

def teste_baixa():
    e = novo()
    baixa_estoque(e, "cimento", 2)
    assert e["cimento"]["qtd"] == 3
    assert insumos_em_alerta(e) == ["cimento"]    # qtd == minimo
```

Executando os testes

```
def teste_baixa_insuficiente():
    try:
        baixa_estoque(novo(), "cimento", 99)
        assert False, "deveria levantar ValueError"
    except ValueError:
        pass

if __name__ == "__main__":
    teste_custo_total()
    teste_baixa()
    teste_baixa_insuficiente()
    print("Todos os testes passaram!")
```

- Rode `python testes.py` após cada alteração
- Para projetos maiores, existem `unittest` (biblioteca padrão) e `pytest` (`pip install pytest`): descobrem e executam todas as funções `test_*` e mostram relatório
- Testes escritos viram a **rede de segurança** das refatorações

Casos extremos: o que testar?

Tipo	Exemplos
Vazio	lista/dicionário/arquivo vazio; texto sem nada
Limites	zero, um, o menor e o maior valor permitido, igual ao mínimo
Inválidos	texto onde se espera número; número negativo; opção inexistente
Tamanho	um só elemento; milhares de linhas
Repetição	item duplicado; cadastrar duas vezes o mesmo nome
Arquivos	inexistente; sem permissão; linha incompleta; acentos
Numéricos	divisão por zero; arredondamento de `float`

Ferramentas de qualidade

Estilo (Aula 24)

```
pylint programa.py
yapf -i programa.py
black programa.py
```

Revisão

- Leia o código do colega *como um estranho*
- Pergunte: dá para entender sem explicação?
- Troque de lugar: um integrante tenta rodar o programa do outro

Código ruim se apaga

Se um trecho está cada vez mais remendado, reescreva-o do zero (Cap. 9.4), agora que você conhece o problema.

Atividade: sessão de refatoração e testes

1. Escrevam **pelo menos 5 testes automáticos** das funções de regra (inclua 2 casos-limite)
2. Passem o `pylint` (ou `yapf`) no código e tratem os principais avisos
3. Procurem e eliminem **código repetido**
4. Renomeiem variáveis e funções pouco claras; acrescentem **docstrings**
5. Verifiquem que nenhuma função de regra usa `input`/`print`
6. Mostrem ao professor: o que mudou? Os testes continuam passando?

Checklist final antes da apresentação

- ☐ O programa roda do zero com um único comando (e há instruções de uso)?
- ☐ Existem testes e todos passam?
- ☐ Nenhuma função tem mais de 30 linhas ou faz várias coisas?
- ☐ Não há código repetido, números mágicos nem variáveis globais mutáveis?
- ☐ Entradas inválidas e erros de arquivo não derrubam o programa?
- ☐ Nomes, docstrings e comentários estão claros?
- ☐ Os dados de exemplo e os arquivos necessários estão na pasta do projeto?
- ☐ Cada integrante consegue explicar qualquer parte do código?