

TÓPICO 11 - TÉCNICAS DE REFATORAÇÃO

Clean Code - Professor Ramon Venson - SATC 2026.2

Refatoração

Refatoração é o processo de alterar a **estrutura interna** do código **sem alterar o seu comportamento externo**.

O código continua fazendo a mesma coisa — mas fica mais fácil de entender, testar e evoluir.

Refatorar **não é reescrever**: é uma sequência de pequenos passos seguros.

Por que refatorar?

- **Combate code smells** identificados na aula anterior;
- **Facilita a leitura** e reduz o tempo de compreensão;
- **Simplifica a adição** de novas funcionalidades;
- **Reduz o custo de manutenção** ao longo do tempo;
- **Melhora o design** sem parar o desenvolvimento.

Refatorar continuamente é o que evita que o código apodreça.

Quando refatorar?

- Ao **identificar um code smell**;
- **Antes** de adicionar uma funcionalidade (deixe espaço);
- **Ao entender** um código confuso (deixe-o melhor do que encontrou);
- **Durante a revisão** de código;
- Quando uma mudança começa a **doer**.

Quando **não** refatorar

- Quando o código é **descartável** ou vai ser reescrito em breve;
- Sem **rede de segurança** (testes) em trechos críticos;
- Misturado com uma mudança de comportamento grande;
- Em prazos apertados, sem tempo para validar.

Nunca refatore e adicione funcionalidade **ao mesmo tempo**.

Refatoração segura

1. **Testes automatizados** garantem o comportamento;
2. **Passos pequenos** e verificáveis a cada etapa;
3. **Controle de versão** permite desfazer rapidamente;
4. **Uma mudança por vez** (estrutura *ou* comportamento);
5. **Commits pequenos** e descritivos.

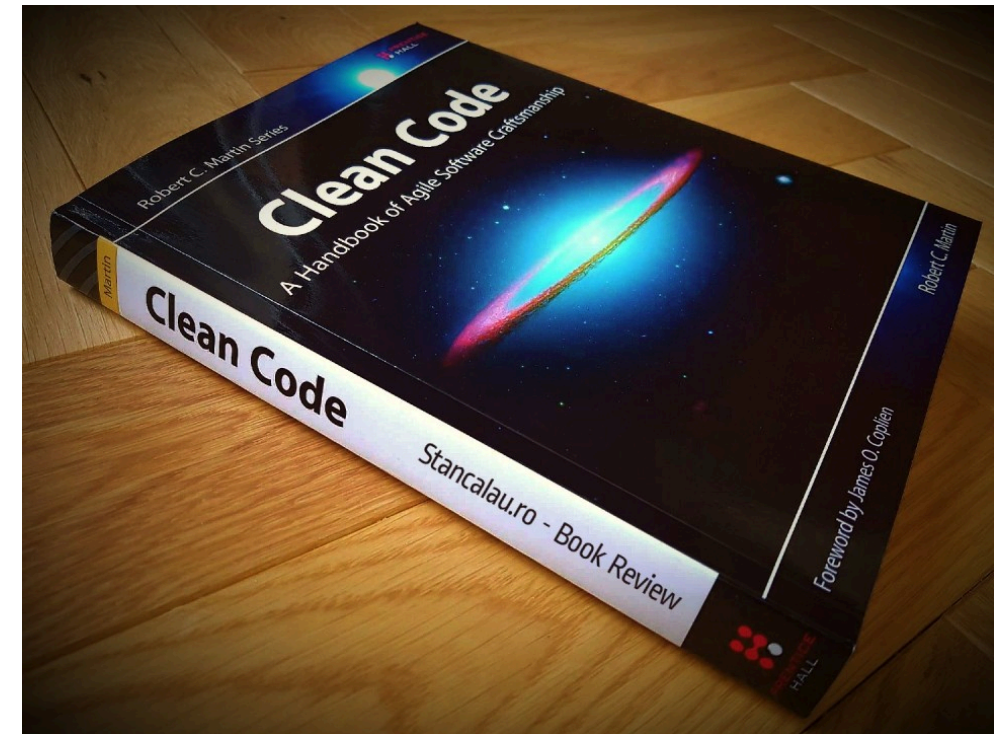
Se você não consegue testar, você não consegue refatorar com segurança.

O catálogo de refatorações

As técnicas de refatoração formam um **catálogo** com nome, motivação e passos.

Vamos organizá-las em seis famílias, seguindo o catálogo do [Refactoring Guru](#):

- Composição de Métodos
- Movendo Recursos entre Objetos
- Organização de Dados
- Simplificação de Condicionais
- Simplificação de Chamadas de Método
- Lidando com Generalização



Da diagnose ao tratamento

Na aula passada vimos os **sintomas** (code smells). Agora veremos os **tratamentos**.

Cada técnica indica **quais cheiros** ela costuma resolver:

Família	Cheiros que combate
Composição de Métodos	Método Longo, Código Duplicado
Movendo Recursos	Inveja de Função, Classe Grande
Organização de Dados	Obsessão por Primitivos, Comandos Switch
Simplificação de Condicionais	Comandos Switch, Condicionais complexas
Simplificação de Chamadas	Lista Longa de Parâmetros
Generalização	Recusa de Herança, Hierarquias Paralelas

O Portal Acadêmico

Continuamos com o **Portal Acadêmico** da aula anterior.

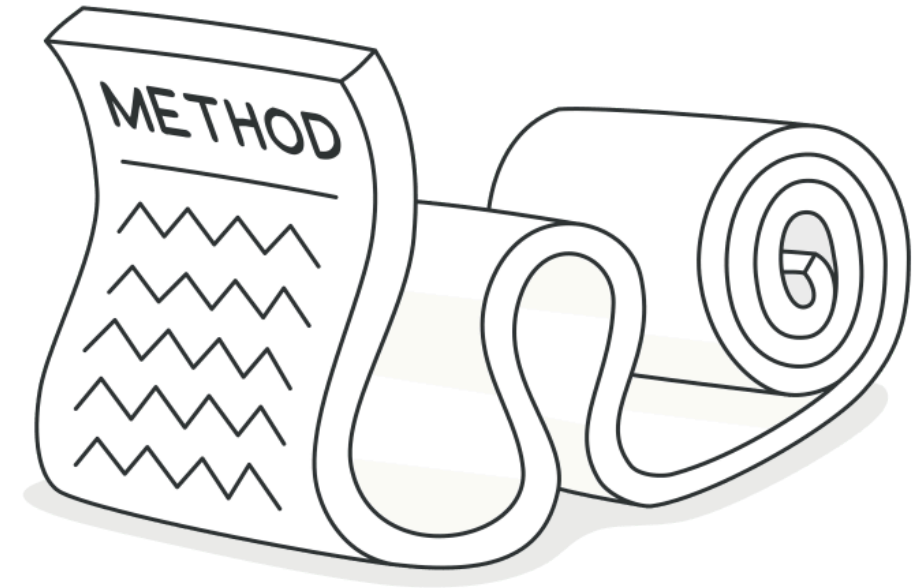
```
class Aluno:
    def __init__(self, nome, cpf, email):
        self.nome = nome
        self.cpf = cpf
        self.email = email

    def calcular_media(self, notas):
        return sum(notas) / len(notas)
```

Os exemplos a seguir mostram como **transformar** os cheiros identificados.

Composição de Métodos

Técnicas que reorganizam métodos para torná-los menores, mais claros e sem duplicação.



Extrair Método (*Extract Method*)

Transforma um trecho de código em um método com nome significativo.

Antes

```
def imprimir_relatorio(aluno):  
    print(f"Aluno: {aluno.nome}")  
    total = 0  
    for n in aluno.notas:  
        total += n  
    print(f"Média: {total / len(aluno.notas):.1f}")
```

Depois

```
def imprimir_relatorio(aluno):  
    print(f"Aluno: {aluno.nome}")  
    print(f"Média: {calcular_media(aluno):.1f}")
```

```
def calcular_media(aluno):  
    return sum(aluno.notas) / len(aluno.notas)
```

Inline Método · Extrair Variável · Inline Variável

Inline Método remove indireção desnecessária:

```
# antes
def media(aluno): return sum(aluno.notas) / len(aluno.notas)
def aprovado(aluno): return media(aluno) >= 7
# depois
def aprovado(aluno): return sum(aluno.notas) / len(aluno.notas) >= 7
```

Extrair Variável dá nome a uma expressão:

```
# antes
if aluno.nota >= 7 and aluno.frequencia >= 0.75: ...
# depois
aprovado = aluno.nota >= 7 and aluno.frequencia >= 0.75
if aprovado: ...
```

Substituir Temp por Consulta · Dividir Variável Temporária

Substituir Temp por Consulta troca variáveis por métodos:

```
# antes
def preco_final(aluno):
    desconto = aluno.mensalidade * 0.1
    return aluno.mensalidade - desconto

# depois
def preco_final(aluno):
    return aluno.mensalidade - desconto(aluno)

def desconto(aluno):
    return aluno.mensalidade * 0.1
```

Dividir Variável Temporária usa uma variável para cada propósito:

```
area = altura * largura # um propósito
perimetro = (altura + largura) * 2 # outro propósito
```

Remover Atribuições a Parâmetros

Evita reutilizar um parâmetro para outro valor dentro do método.

Antes

```
def normalizar(nota):  
    nota = nota / 10  
    return nota
```

Depois

```
def normalizar(nota):  
    return nota / 10
```

Resolve: confusão sobre o valor original do parâmetro.

Substituir Método por Objeto-Método · Substituir Algoritmo

Objeto-Método extrai um método complexo para uma classe própria:

```
class CalculadoraMedia:
    def __init__(self, notas, pesos): ...
    def calcular(self): ...
```

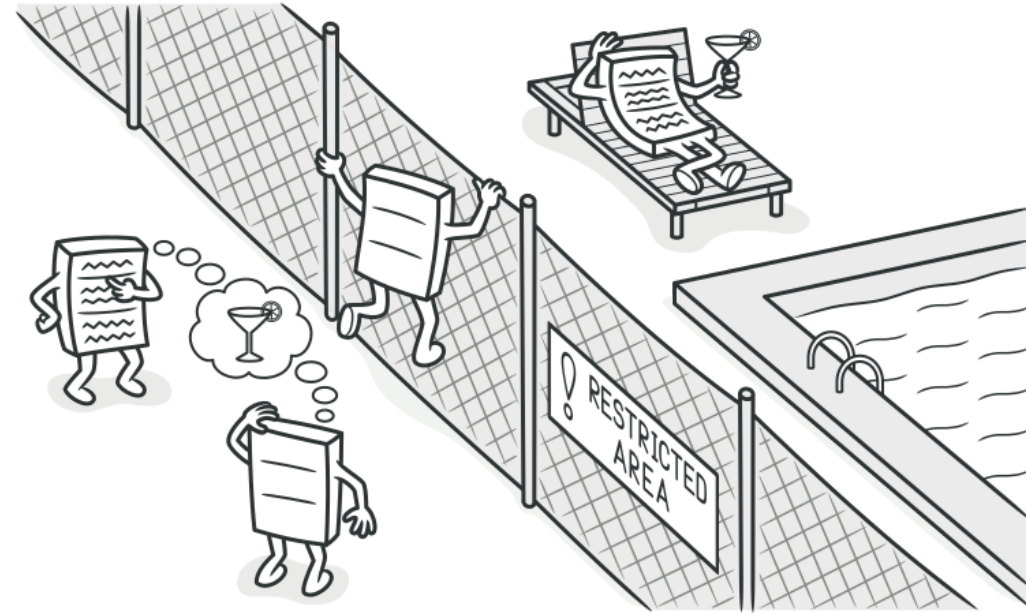
Substituir Algoritmo troca um algoritmo confuso por um mais claro:

```
# antes
def contem(lista, x):
    for i in range(len(lista)):
        if lista[i] == x: return True
    return False

# depois
def contem(lista, x):
    return x in lista
```

Movendo Recursos entre Objetos

Técnicas que movem métodos e campos para a classe onde eles realmente pertencem.



Mover Método · Mover Campo

Mover Método leva o método para quem tem os dados:

```
# antes
class Relatorio:
    def calcular_media(self, aluno):
        return sum(aluno.notas) / len(aluno.notas)

# depois
class Aluno:
    def calcular_media(self):
        return sum(self.notas) / len(self.notas)
```

Mover Campo acompanha o método quando o dado também está no lugar errado.

Resolve: Inveja de Função.

Extrair Classe (*Extract Class*)

Separa responsabilidades de uma classe grande em classes coesas.

Antes

```
class Aluno:
    def __init__(self, nome, rua, numero, cidade, cep):
        self.nome = nome
        self.rua, self.numero = rua, numero
        self.cidade, self.cep = cidade, cep
```

Depois

```
class Endereco:
    def __init__(self, rua, numero, cidade, cep): ...

class Aluno:
    def __init__(self, nome, endereco):
        self.nome = nome
        self.endereco = endereco
```

Inline Classe (*Inline Class*)

O inverso: incorpora uma classe que faz pouco na classe que a usa.

Antes

```
class Nota:
    def __init__(self, valor):
        self.valor = valor
# Aluno usa apenas Nota.valor
```

Depois

```
class Aluno:
    def __init__(self, nome):
        self.nome = nome
        self.notas = [] # valores diretamente
```

Ocultar Delegado (*Hide Delegate*)

Cria um método intermediário para esconder a cadeia de chamadas.

Antes

```
turma = aluno.get_matricula().get_turma()
```

Depois

```
class Aluno:
    def get_turma(self):
        return self.matricula.get_turma()

turma = aluno.get_turma()
```

Resolve: Cadeia de Mensagens.

Remover Homem do Meio (*Remove Middle Man*)

Elimina intermediários que só repassam chamadas.

Antes

```
class Coordenacao:  
    def matricular(self, aluno, turma):  
        return self.secretaria.matricular(aluno, turma)
```

Depois

```
# cliente chama a secretaria diretamente  
secretaria.matricular(aluno, turma)
```

Resolve: Homem do Meio.

Introduzir Método Estranho · Introduzir Extensão Local

Método Estranho: um auxiliar para a classe de biblioteca que você não pode alterar.

```
def proximo_dia_letivo(data):  
    return data + timedelta(days=1)
```

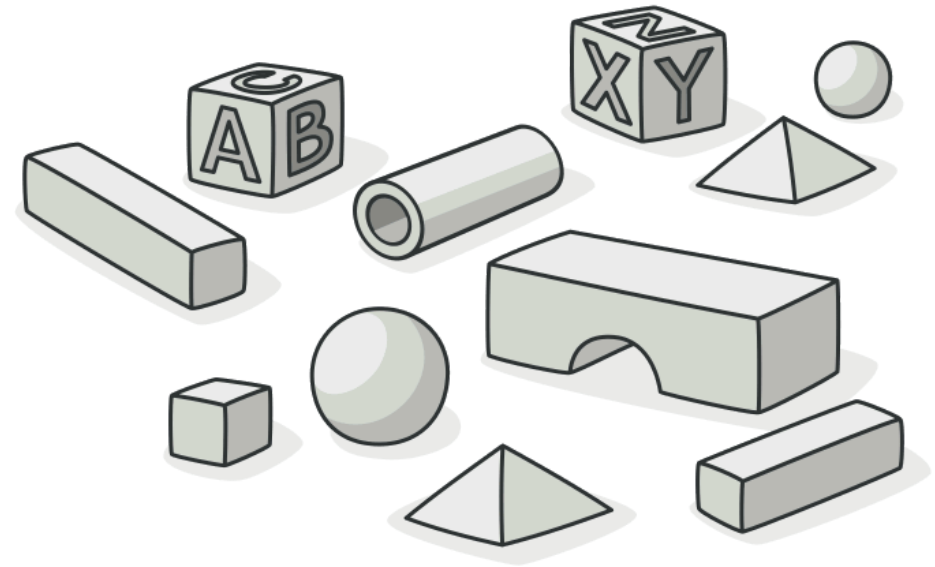
Extensão Local: uma subclasse ou *wrapper* que agrupa esses auxiliares.

```
class DataLetiva(date):  
    def proximo_dia_letivo(self):  
        return self + timedelta(days=1)
```

Resolve: Classe de Biblioteca Incompleta.

Organização de Dados

Técnicas que substituem valores primitivos por objetos e controlam o acesso aos dados.



Encapsular Campo · Autoencapsular Campo

Encapsular Campo esconde o acesso direto aos dados:

```
# antes
aluno.nota = -5    # qualquer valor!
# depois
class Aluno:
    def set_nota(self, valor):
        if valor < 0: raise ValueError
        self._nota = valor
```

Autoencapsular Campo faz a própria classe acessar seus campos via métodos, permitindo mudanças futuras.

Encapsular Coleção (*Encapsulate Collection*)

Impede que o cliente altere a coleção interna sem controle.

Antes

```
aluno.notas.append(11) # lista exposta
```

Depois

```
class Aluno:
    def adicionar_nota(self, nota):
        if 0 <= nota <= 10:
            self._notas.append(nota)

    def get_notas(self):
        return list(self._notas) # cópia
```

Substituir Valor de Dados por Objeto

Troca um primitivo por um objeto com validação e comportamento.

Antes

```
private String cpf;    // "123.456.789-00" ou qualquer coisa
```

Depois

```
public class Cpf {  
    private final String valor;  
    public Cpf(String valor) {  
        if (!valida(valor)) throw new IllegalArgumentException();  
        this.valor = valor;  
    }  
}
```

Substituir Código de Tipo (*Replace Type Code*)

Substitui um código por **Classe**, **Subclasses** ou **State/Strategy**.

Antes

```
def calcular_conceito(media, curso):  
    if curso == "engenharia":  
        return "A" if media >= 9 else "B" if media >= 7 else "C"  
    elif curso == "direito":  
        return "A" if media >= 8.5 else "B" if media >= 6 else "C"
```

Depois

```
class Curso(ABC):  
    @abstractmethod  
    def conceito(self, media): ...
```

```
class Engenharia(Curso):  
    def conceito(self, media):  
        return "A" if media >= 9 else "B" if media >= 7 else "C"
```

Substituir Array por Objeto

Um array com posições fixas vira um objeto com campos nomeados.

Antes

```
dados = ["Ana", "123.456.789-00", "Engenharia"]  
nome, cpf, curso = dados[0], dados[1], dados[2]
```

Depois

```
aluno = Aluno(nome="Ana", cpf="123.456.789-00", curso="Engenharia")
```

Substituir Número Mágico por Constante

Dá nome a valores soltos no código.

Antes

```
if media >= 7:  
    ...
```

Depois

```
MEDIA_APROVACAO = 7  
  
if media >= MEDIA_APROVACAO:  
    ...
```

Substituir Valor por Referência · Referência por Valor

Valor por Referência: vários objetos iguais passam a compartilhar uma única instância.

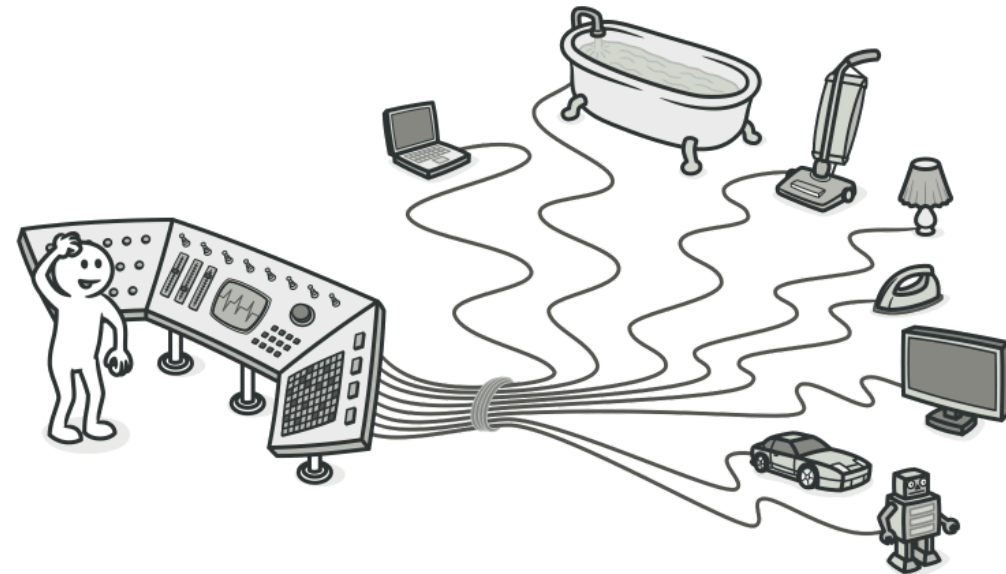
Referência por Valor: objetos pequenos e imutáveis viram valores copiáveis.

```
# valor -> referência: alunos do mesmo curso  
# compartilham o mesmo objeto Curso  
aluno_a.curso = curso_engenharia  
aluno_b.curso = curso_engenharia
```

Resolve: duplicação de dados e comparações por identidade equivocadas.

Simplificação de Condicionais

Técnicas que tornam expressões condicionais complexas mais simples de ler.



Decompor Condicional (*Decompose Conditional*)

Extrai a condição e os ramos para métodos com nomes claros.

Antes

```
if hoje.weekday() >= 5 or hoje in feriados:
    valor = base * 0
else:
    valor = base * 1.2
```

Depois

```
if e_dia_nao_letivo(hoje):
    valor = sem_aula(base)
else:
    valor = com_aula(base)
```

Consolidar Condicional · Consolidar Fragmentos Duplicados

Consolidar Condicional: junta condições com o mesmo resultado.

```
# antes
if aluno.bolsista: return 0
if aluno.funcionario: return 0
# depois
if aluno.bolsista or aluno.funcionario: return 0
```

Consolidar Fragmentos Duplicados: move o código comum para fora dos ramos.

```
if aprovado:
    registrar(aluno)
    emitir_certificado(aluno)
else:
    registrar(aluno)
    emitir_pendencia(aluno)
```

Remover Flag de Controle (*Remove Control Flag*)

Substitui variáveis de controle por `break`, `return` ou `continue`.

Antes

```
encontrado = False
for aluno in alunos:
    if aluno.id == alvo:
        encontrado = True
        break
```

Depois

```
for aluno in alunos:
    if aluno.id == alvo:
        return aluno
```

Substituir Condicional Aninhada por Guard Clauses

Usa retornos antecipados para eliminar aninhamento.

Antes

```
def matricular(aluno):  
    if aluno.ativo:  
        if aluno.pendencias == 0:  
            if vagas_disponiveis():  
                efetivar(aluno)
```

Depois

```
def matricular(aluno):  
    if not aluno.ativo: return  
    if aluno.pendencias > 0: return  
    if not vagas_disponiveis(): return  
    efetivar(aluno)
```

Substituir Condicional por Polimorfismo

Cada variação de comportamento vira uma classe.

Antes

```
def exportar(formato):  
    if formato == "pdf": ...  
    elif formato == "csv": ...  
    elif formato == "html": ...
```

Depois

```
class Exportador(ABC):  
    @abstractmethod  
    def exportar(self): ...  
  
class PdfExportador(Exportador): ...  
class CsvExportador(Exportador): ...
```

Introduzir Null Object

Um objeto "vazio" substitui verificações repetidas de `None`.

Antes

```
if aluno is None:
    print("Aluno não encontrado")
else:
    print(aluno.nome)
```

Depois

```
class AlunoNulo:
    nome = "Aluno não encontrado"

aluno = buscar(aluno_id) or AlunoNulo()
print(aluno.nome)
```

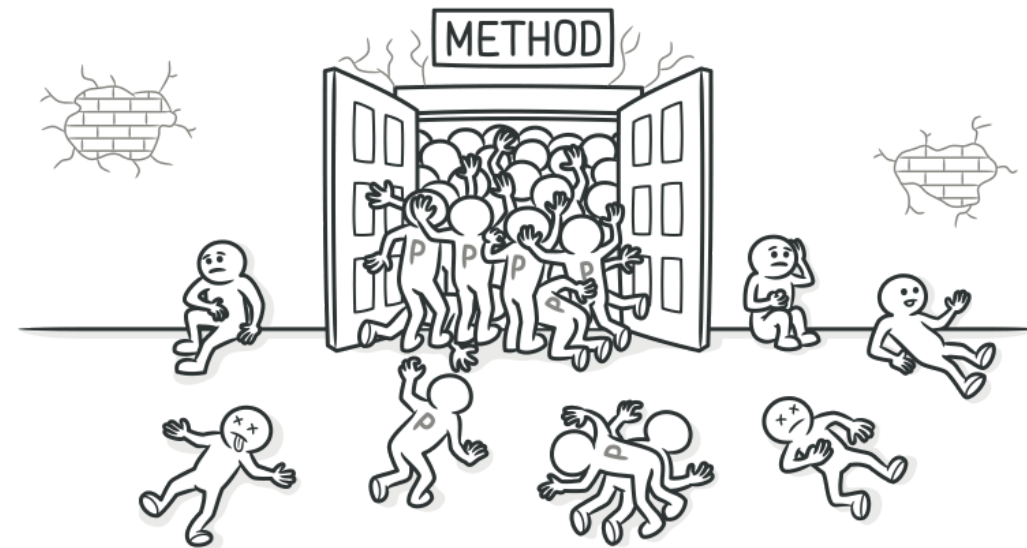
Introduzir Asserção

Documenta uma suposição que **deve** ser verdadeira.

```
def calcular_media(notas):  
    assert len(notas) > 0, "a lista de notas não pode ser vazia"  
    return sum(notas) / len(notas)
```

Simplificação de Chamadas de Método

Técnicas que tornam as interfaces entre objetos mais simples e claras.



Renomear Método (*Rename Method*)

Um bom nome elimina a necessidade de explicações.

Antes

```
def proc(a): ...
```

Depois

```
def calcular_media(aluno): ...
```

Adicionar Parâmetro · Remover Parâmetro

Adicionar: quando o método precisa de mais informação.

Remover: quando um parâmetro não é mais usado.

```
# antes: parâmetro ignorado
def matricular(aluno, turma, ano_letivo):
    return aluno.matricular(turma)
# depois
def matricular(aluno, turma):
    return aluno.matricular(turma)
```

Resolve: interfaces confusas e Generalidade Especulativa.

Separar Consulta de Modificador

Um método que responde e altera estado ao mesmo tempo é confuso.

Antes

```
def proxima_matricula(self):  
    self.contador += 1  
    return self.contador
```

Depois

```
def get_matricula(self):  
    return self.contador  
  
def incrementar(self):  
    self.contador += 1
```

Parametrizar Método · Substituir Parâmetro por Métodos Explícitos

Parametrizar: vários métodos iguais viram um só com parâmetro.

```
def aumentar_por(self, fator): ...
```

Métodos Explícitos: um parâmetro *flag* vira métodos separados.

```
# antes
def pagar(self, bolsista: bool): ...
# depois
def pagar_integral(self): ...
def pagar_com_desconto(self): ...
```

Preservar Objeto Inteiro

Passe o objeto em vez de vários de seus campos.

Antes

```
def calcular_distancia(rua, numero, cidade, cep): ...  
    calcular_distancia(a.rua, a.numero, a.cidade, a.cep)
```

Depois

```
def calcular_distancia(endereco): ...  
    calcular_distancia(aluno.endereco)
```

Resolve: Lista Longa de Parâmetros, Agrupamentos de Dados.

Substituir Parâmetro por Chamada de Método

O método pode obter o valor sozinho.

Antes

```
def boletim(aluno, notas): ...  
    boletim(aluno, aluno.notas)
```

Depois

```
def boletim(aluno):  
    return ...aluno.notas...
```

Introduzir Objeto de Parâmetro

Agrupa vários parâmetros que sempre andam juntos.

Antes

```
def buscar_alunos(nome, curso, semestre, status, cidade, estado):  
    ...
```

Depois

```
class FiltroAluno:  
    def __init__(self, nome, curso, semestre, status, cidade, estado): ...  
  
def buscar_alunos(filtro: FiltroAluno):  
    ...
```

Resolve: Lista Longa de Parâmetros.

Remover Método de Ajuste · Ocultar Método

Remover Método de Ajuste: elimina *setters* que não deveriam existir.

```
# antes  
aluno.set_matricula("2024-1")  
# depois: definido apenas na criação  
Aluno(nome, matricula="2024-1")
```

Ocultar Método: restringe a visibilidade de métodos internos.

Resolve: Intimidade Inapropriada.

Substituir Construtor por Factory Method

Fábricas expressam melhor as variações de criação.

Antes

```
aluno = Aluno(nome, tipo="EAD")
```

Depois

```
class Aluno:  
    @staticmethod  
    def criar_ead(nome):  
        return Aluno(nome, modalidade="ead")
```

Substituir Código de Erro por Exceção · Exceção por Teste

Código de erro -> exceção: erros deixam de passar silenciosamente.

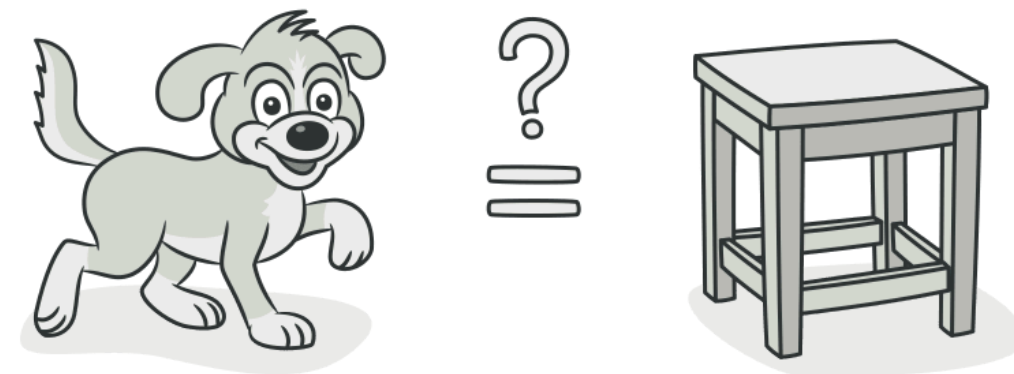
```
# antes
def buscar(id): return None # -1?
# depois
def buscar(id):
    if id not in self.alunos: raise AlunoNaoEncontrado
```

Exceção -> teste: não use exceções para fluxo esperado.

```
# antes
try: valor = dados["nota"]
except KeyError: valor = 0
# depois
valor = dados.get("nota", 0)
```

Lidando com Generalização

Técnicas que reorganizam hierarquias de herança e a relação entre classes.



Pull Up Campo · Pull Up Método · Pull Up Corpo de Construtor

Move membros duplicados das subclasses para a superclasse.

Antes

```
class AlunoPresencial:
    def __init__(self, nome): self.nome = nome

class AlunoEAD:
    def __init__(self, nome): self.nome = nome
```

Depois

```
class Aluno:
    def __init__(self, nome): self.nome = nome

class AlunoPresencial(Aluno): ...
class AlunoEAD(Aluno): ...
```

Push Down Método · Push Down Campo

Move membros da superclasse para a(s) subclasse(s) que realmente os usam.

```
# antes: só AlunoEAD usa
class Aluno:
    def acessar_ava(self): ...

# depois
class AlunoEAD(Aluno):
    def acessar_ava(self): ...
```

Resolve: Recusa de Herança.

Extrair Subclasse · Extrair Superclasse

Extrair Subclasse: uma característica opcional vira subclasse.

```
class AlunoBolsista(Aluno): ...
```

Extrair Superclasse: classes com código comum ganham uma base compartilhada.

Resolve: Hierarquias mal formadas, Código Duplicado.

Extrair Interface (*Extract Interface*)

Extrai uma interface comum de classes com métodos semelhantes.

Antes

```
class AlunoPresencial:
    def registrar_presenca_aula(self, aula): ...
class AlunoEAD:
    def marcar_acesso_online(self, aula): ...
```

Depois

```
class Matriculavel(ABC):
    @abstractmethod
    def registrar_presenca(self, aula): ...
```

Resolve: Classes Alternativas com Interfaces Diferentes.

Colapsar Hierarquia (*Collapse Hierarchy*)

Funde subclasse e superclasse quando não há diferença real.

```
# antes
class Aluno: ...
class AlunoAtivo(Aluno): ... # sem diferença
# depois
class Aluno: ... # apenas
```

Resolve: Classe Preguiçosa, Generalidade Especulativa.

Form Template Method

Define o esqueleto do algoritmo na base e delega passos às subclasses.

```
class ProcessadorNota(ABC):  
    def processar(self):  
        self.validar()  
        self.calcular()  
        self.persistir()  
  
    @abstractmethod  
    def calcular(self): ...
```

Resolve: Código Duplicado com variações.

Substituir Herança por Delegação

Quando a subclasse usa só parte do comportamento, prefira composição.

Antes

```
class AlunoBolsista(Aluno):  
    def pagar_mensalidade(self):  
        raise NotImplementedError
```

Depois

```
class Aluno:  
    def __init__(self, nome, plano_pagamento):  
        self.plano = plano_pagamento # delegação
```

Resolve: Recusa de Herança.

Substituir Delegação por Herança

O inverso: quando a delegação cobre toda a interface, herdar simplifica.

```
# antes
class Aluno:
    def __init__(self, pessoa):
        self.pessoa = pessoa
    def get_nome(self):
        return self.pessoa.get_nome()

# depois
class Aluno(Pessoa): ...
```

Resolve: Homem do Meio.

Boas práticas ao refatorar

- **Pequenos passos:** uma transformação por vez, testando sempre;
- **Testes antes de refatorar:** eles definem o comportamento esperado;
- **Nomes melhores** a cada oportunidade;
- **Sem misturar** refatoração com novas funcionalidades;
- **Commits descritivos** para facilitar a reversão;
- **Refatoração contínua:** não deixe para uma "grande faxina".

Cuidado: cheiros de refatoração

A própria refatoração pode introduzir problemas:

- Refatorar **sem testes** gera bugs silenciosos;
- **Passos grandes** tornam difícil encontrar o erro;
- **Abstrações prematuras** criam Generalidade Especulativa;
- Refatorações que **nunca terminam** atrasam a entrega.

O objetivo é um código melhor, não a refatoração pela refatoração.

Ligação com o projeto

No **projeto de refatoração**, vocês vão:

1. Identificar os **code smells** (tópico 10);
2. Escolher as **técnicas** adequadas (tópico 11);
3. Aplicar as transformações **preservando o comportamento**;
4. Justificar as escolhas com base nos princípios estudados.

A qualidade do código é medida pela facilidade de mudá-lo.

Material de Apoio

- [Refactoring Guru — Técnicas de Refatoração](#)
- [Martin Fowler — Refactoring \(2ª ed.\)](#)
- [Refactoring.com — Catálogo](#)
- [SourceMaking — Refactoring](#)