

TÓPICO 10 - CODE SMELLS

Clean Code - Professor Ramon Venson - SATC 2026.2

Code Smells

Code smells (ou "maus cheiros") são **sinais** de que algo pode estar errado no código.

Um code smell **não é um erro**: o programa pode compilar e funcionar perfeitamente.

Ainda assim, ele indica que o design pode ser melhorado.

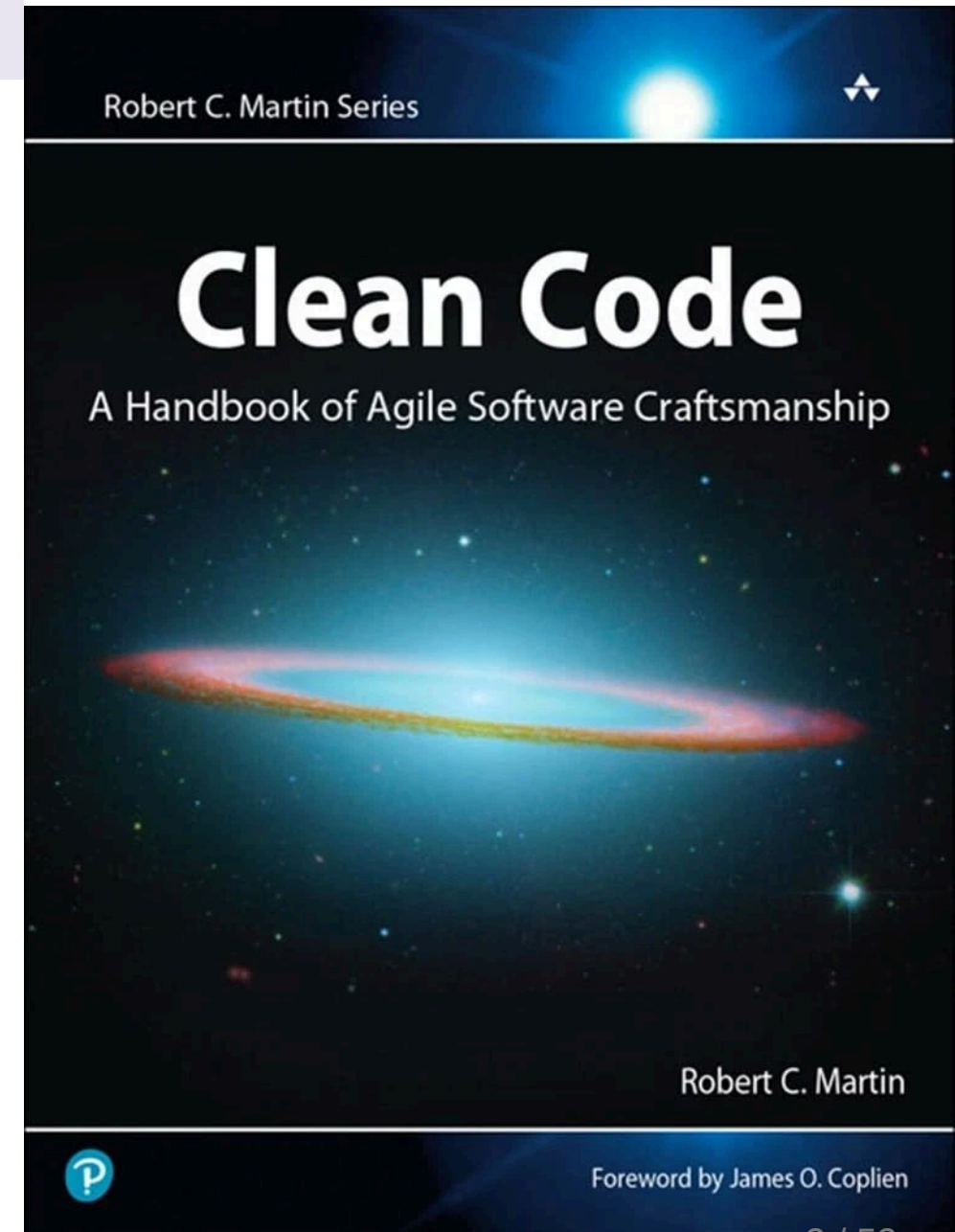
| O cheiro é o sintoma. O diagnóstico e o tratamento exigem análise.

Sintoma, não diagnóstico

Na medicina, um **sintoma** (febre, dor) não é a doença: é um indício que orienta o diagnóstico.

Com código acontece o mesmo:

- O cheiro aponta **onde olhar**;
- O diagnóstico exige **entender o contexto**;
- O tratamento vem depois — e é o assunto do **próximo tópico**.



As cinco categorias

Categoria	Descrição	Cheiros
Inchaços (<i>Bloaters</i>)	Código que cresceu demais e ficou difícil de trabalhar.	5
Abusos da OO (<i>OO Abusers</i>)	Maus usos ou aplicações incompletas de orientação a objetos.	4
Impedidores de Mudança (<i>Change Preventers</i>)	Mudar um ponto força mudanças em vários outros.	3
Dispensáveis (<i>Dispensables</i>)	Elementos desnecessários que poderiam ser removidos.	6
Acopladores (<i>Couplers</i>)	Acoplamento excessivo entre classes.	5

Por que isso importa?

Code smells encarecem a manutenção:

- **Legibilidade:** o código fica difícil de entender;
- **Mudança:** pequenas alterações viram grandes tarefas;
- **Testes:** fica mais difícil isolar e testar comportamentos;
- **Colaboração:** o time perde tempo decifrando o código;
- **Evolução:** novas funcionalidades quebram o que já existia.

O custo de mudar um software cresce com o tempo. Code smells aceleram esse crescimento.

O Portal Acadêmico

Para dar contexto, vamos usar um mesmo sistema em todos os exemplos: o **Portal Acadêmico**.

Ele gerencia:

- **Alunos, professores e disciplinas;**
- **Matrículas, turmas e notas;**
- **Boletins, históricos e relatórios;**
- **Notificações e exportação de dados.**

Um recorte do Portal

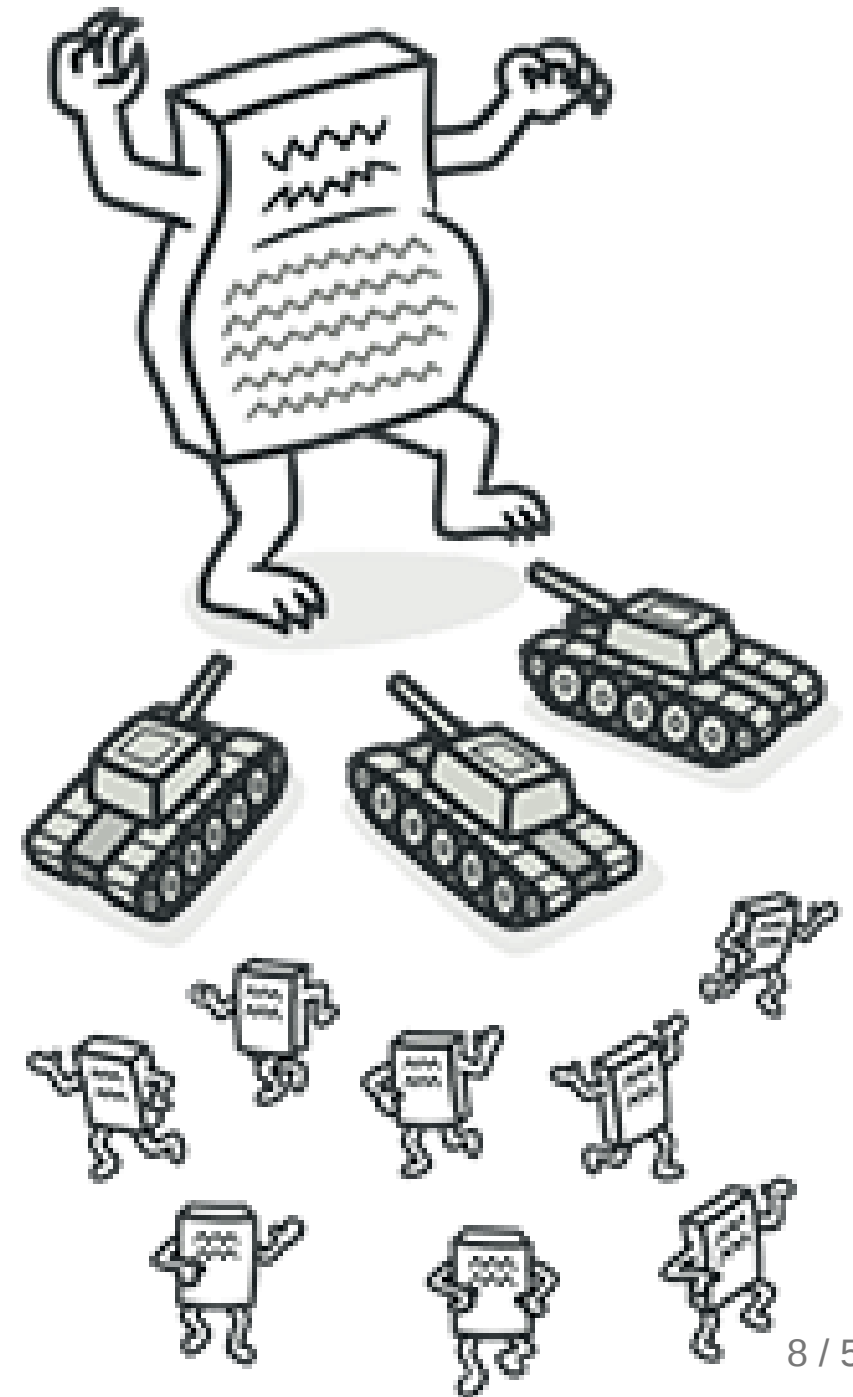
```
class Aluno:
    def __init__(self, nome, cpf, email):
        self.nome = nome
        self.cpf = cpf
        self.email = email

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

Esse recorte é **razoável**. Ao longo do tópico, vamos vê-lo se degradar de diferentes formas.

Inchaços (Bloaters)

Códigos, métodos e classes que cresceram a proporções tão gigantescas que se tornam difíceis de trabalhar.

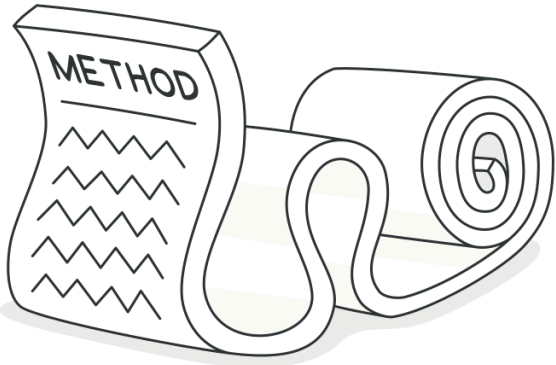


Mapa dos Inchaços

Grupo	Cheiro	Sinal de alerta	Onde dói
A	Método Longo	dezenas de linhas, muitos níveis	difícil de ler e testar
A	Classe Grande	muitas responsabilidades	difícil de reutilizar
B	Obsessão por Primitivos	<code>str</code> / <code>int</code> para conceitos ricos	validação espalhada
B	Lista Longa de Parâmetros	muitos parâmetros	chamadas confusas
B	Agrupamentos de Dados	os mesmos dados sempre juntos	mudanças em cascata

Método Longo (*Long Method*)

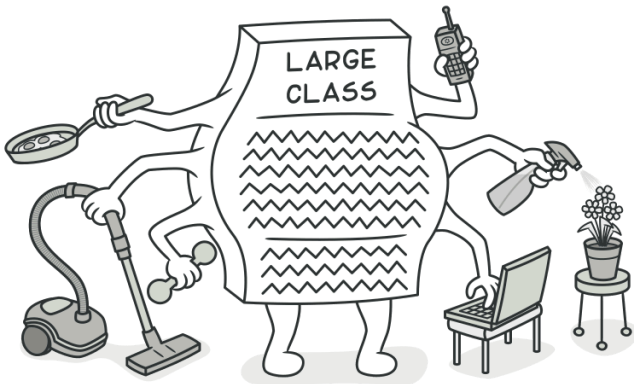
Um método contém linhas demais e acumula responsabilidades.



```
def gerar_boletim(aluno_id):  
    aluno = db.buscar_aluno(aluno_id)  
    disciplinas = db.buscar_disciplinas(aluno_id)  
    linhas = []  
    for d in disciplinas:  
        notas = db.buscar_notas(aluno_id, d.id)  
        linhas.append(f"{d.nome}: {sum(notas) / len(notas):.1f}")  
    texto = "\n".join(linhas)  
    email.enviar(aluno.email, "Boletim", texto)  
    db.salvar_historico(aluno_id, texto)  
    return texto
```

Classe Grande (*Large Class*)

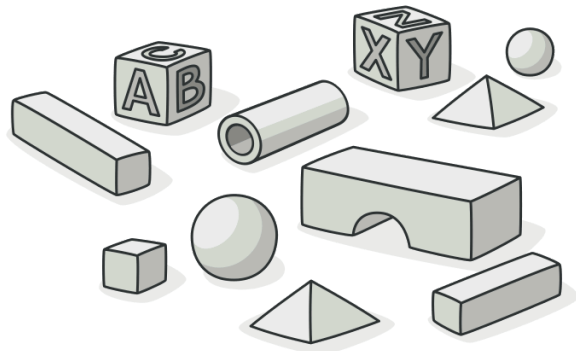
Uma classe acumula responsabilidades demais e vira um "Deus".



```
class Aluno:
    def matricular(self, disciplina): ...
    def cancelar_matricula(self, disciplina): ...
    def calcular_media(self): ...
    def gerar_boletim(self): ...
    def enviar_email(self, assunto, corpo): ...
    def enviar_sms(self, texto): ...
    def alterar_senha(self, senha): ...
    def registrar_presenca(self, aula): ...
    def exportar_dados(self, formato): ...
```

Obsessão por Primitivos (*Primitive Obsession*)

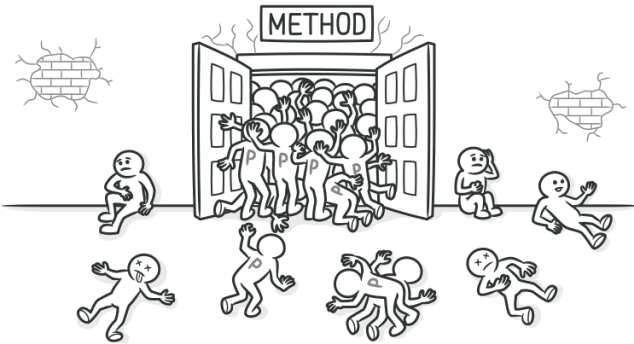
Uso excessivo de tipos primitivos no lugar de objetos específicos.



```
public class Aluno {  
    private String nome;  
    private String cpf;  
    private String email;  
    private String telefone;  
    private double nota1;  
    private double nota2;  
    private double nota3;  
    private double nota4;  
}
```

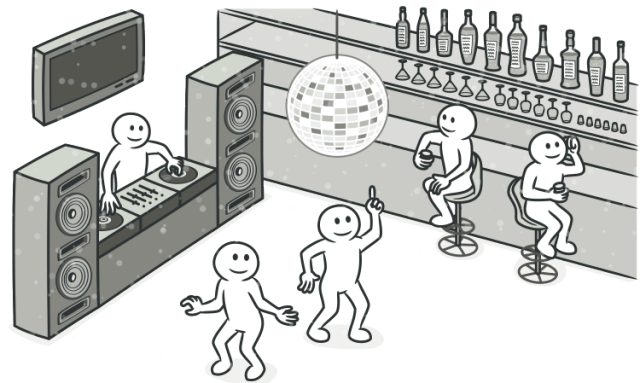
Lista Longa de Parâmetros (*Long Parameter List*)

Métodos que exigem parâmetros demais, difíceis de lembrar e usar.



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

Quem chama precisa acertar a ordem de oito valores diferentes.



Agrupamentos de Dados (*Data Clumps*)

Grupos de variáveis que sempre aparecem juntas.

```
def cadastrar_aluno(nome, rua, numero, bairro, cidade, estado, cep):  
    ...  
  
def atualizar_endereco(aluno, rua, numero, bairro, cidade, estado, cep):  
    ...
```

O grupo de endereço se repete por toda a aplicação.

Checkpoint — Qual o cheiro?

```
def fechar_semestre(turma_id):  
    turma = db.buscar_turma(turma_id)  
    for aluno in turma.alunos:  
        notas = db.buscar_notas(aluno.id)  
        media = sum(notas) / len(notas)  
        if media >= 7:  
            db.aprovar(aluno.id)  
            email.enviar(aluno.email, "Aprovado", f"Média {media}")  
        else:  
            db.reprovar(aluno.id)  
            email.enviar(aluno.email, "Reprovado", f"Média {media}")  
    db.salvar_log("semestre fechado")
```

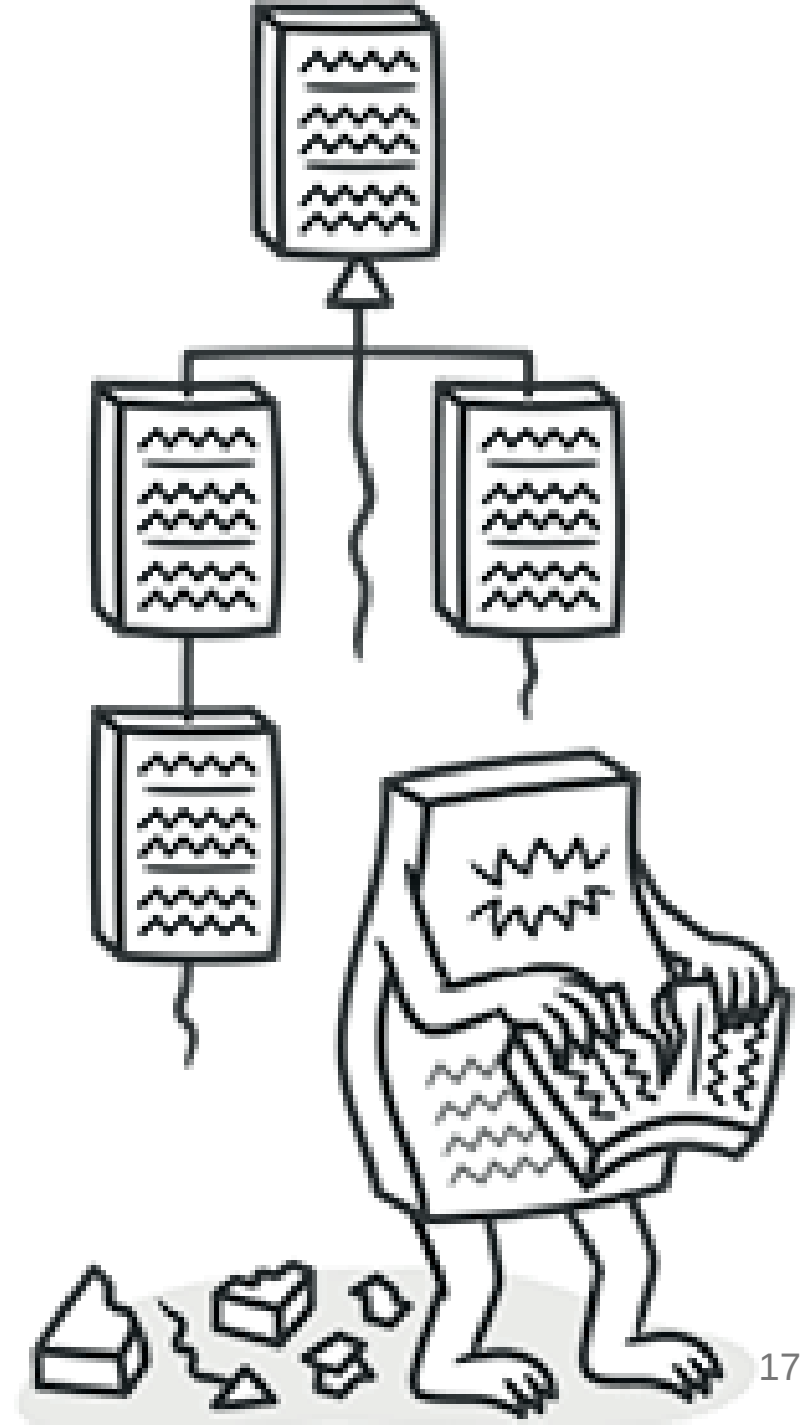
Checkpoint — Resposta

Método Longo: o método faz coisas demais (buscar, calcular, decidir, notificar e registrar).

Há também um indício de **Mudança Divergente:** regra de negócio, notificação e persistência convivem no mesmo lugar.

Abusos da Orientação a Objetos

Maus usos ou aplicações incompletas de princípios da orientação a objetos.

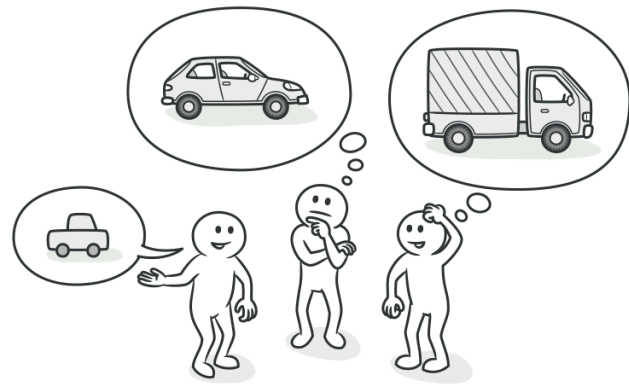


Mapa dos Abusos da OO

Grupo	Cheiro	Sinal de alerta	Onde dói
C	Classes Alternativas com Interfaces Diferentes	irmãs com métodos de nomes diferentes	uso inconsistente
C	Recusa de Herança	subclasse ignora o que herda	hierarquia enganosa
D	Comandos Switch	<code>if / elif</code> por tipo	novo tipo exige editar tudo
D	Campo Temporário	atributo usado só às vezes	estado inconsistente

Classes Alternativas com Interfaces Diferentes

Classes fazem coisas semelhantes, mas com interfaces diferentes.

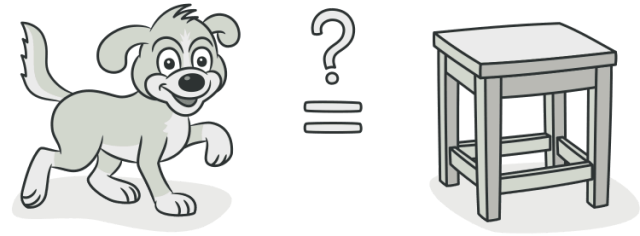


```
class AlunoPresencial:  
    def registrar_presenca_aula(self, aula): ...  
  
class AlunoEAD:  
    def marcar_acesso_online(self, aula): ...
```

Quem usa precisa tratar cada classe de um jeito.

Recusa de Herança (*Refused Bequest*)

Uma subclasse herda comportamento que não utiliza.



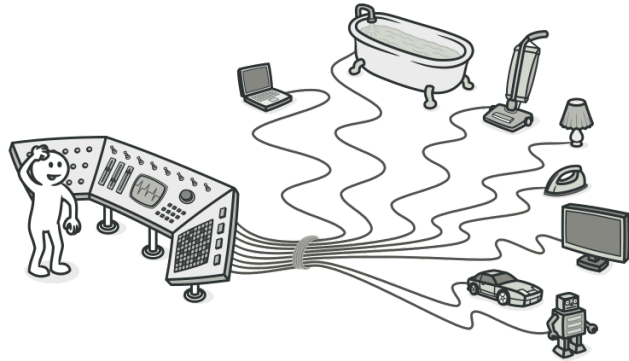
```
class Aluno:
    def pagar_mensalidade(self): ...
    def calcular_desconto(self): ...

class AlunoBolsista(Aluno):
    def pagar_mensalidade(self):
        raise NotImplementedError("Bolsista não paga")
```

A hierarquia promete um comportamento que não se aplica.

Comandos Switch (*Switch Statements*)

Uso excessivo de `switch` ou `if / elif` para alterar comportamento.

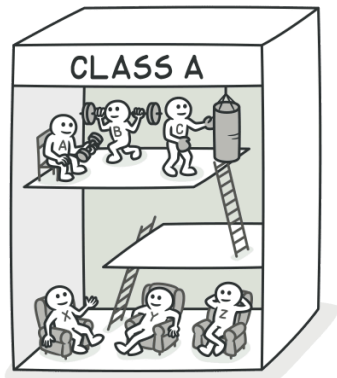


```
def calcular_conceito(media, curso):  
    if curso == "engenharia":  
        return "A" if media >= 9 else "B" if media >= 7 else "C"  
    elif curso == "direito":  
        return "A" if media >= 8.5 else "B" if media >= 6 else "C"  
    elif curso == "medicina":  
        return "A" if media >= 9.5 else "B" if media >= 8 else "C"
```

Cada novo curso exige mais um ramo.

Campo Temporário (*Temporary Field*)

Campo usado apenas em situações específicas.



```
class Relatorio:
    def __init__(self):
        self.aluno = None
        self.media = None
        self.turma_em_fechamento = None # só usado no fim do semestre

    def fechar_semestre(self, turma):
        self.turma_em_fechamento = turma
        ...
```

Fora desse momento, o campo fica sem sentido.

Checkpoint — Qual o cheiro?

```
class AlunoRegular(Aluno):  
    def cobrar_mensalidade(self):  
        return self.valor_mensalidade  
  
class AlunoBolsista(Aluno):  
    def cobrar_mensalidade(self):  
        raise Exception("Bolsista não paga mensalidade")
```

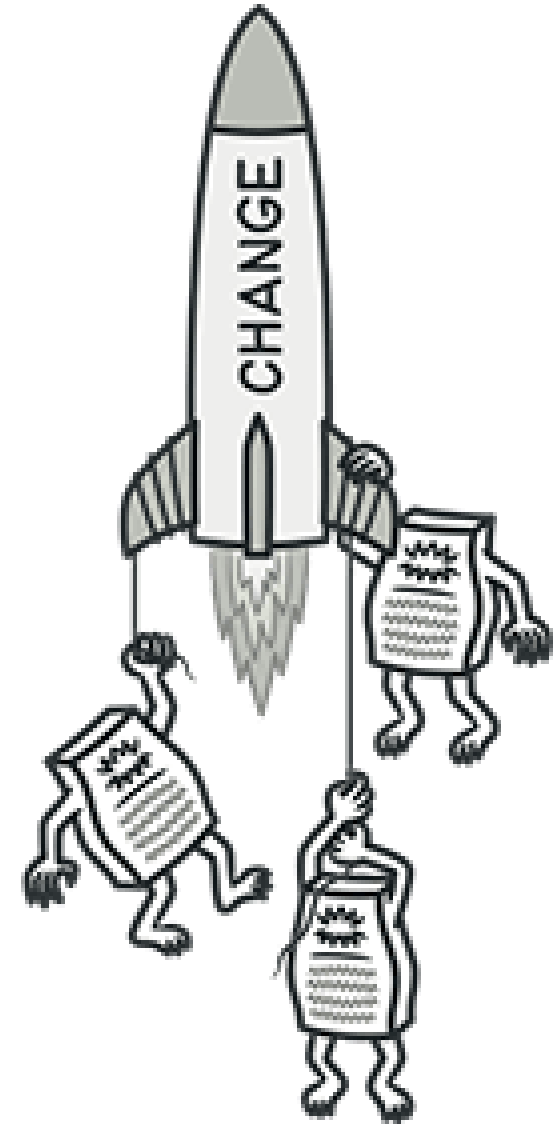
Checkpoint — Resposta

Recusa de Herança: `AlunoBolsista` é obrigado a implementar um comportamento que não faz sentido para ele.

O sinal é o método que existe só para **negar** a herança.

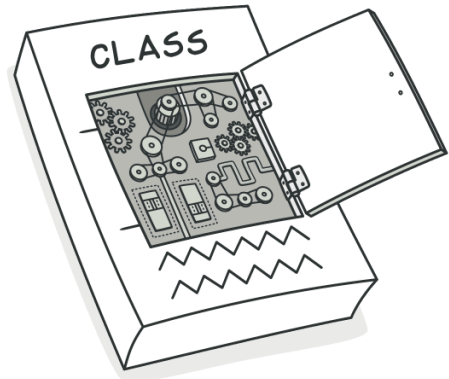
Impedidores de Mudança

Mudanças em um ponto do sistema forçam mudanças em vários outros lugares.



Mapa dos Impedidores de Mudança

Grupo	Cheiro	Sinal de alerta	Onde dói
E	Mudança Divergente	uma classe muda por vários motivos	manutenção arriscada
E	Cirurgia de Espingarda	uma mudança toca vários lugares	erro fácil de cometer
E	Hierarquias de Herança Paralelas	criar subclasse em duas hierarquias	explosão de classes



Mudança Divergente (*Divergent Change*)

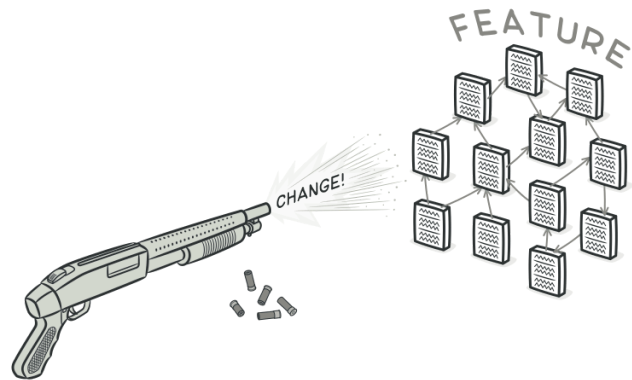
Uma mesma classe precisa mudar por motivos diferentes.

```
class Secretaria:  
    def formatar_boletim(self): ...    # muda por regra de apresentação  
    def calcular_media(self): ...     # muda por regra acadêmica  
    def salvar_no_banco(self): ...    # muda por schema/infraestrutura
```

Três motivos independentes de mudança no mesmo lugar.

Cirurgia de Espingarda (*Shotgun Surgery*)

Uma pequena mudança exige alterações em vários lugares diferentes.

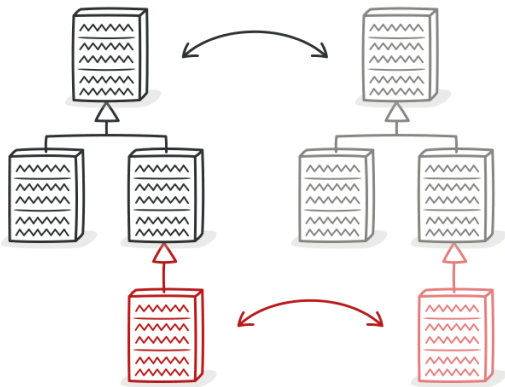


```
def aprovado(notas):  
    return sum(notas) / len(notas) >= 7  
  
def bolsista(notas):  
    return sum(notas) / len(notas) >= 8  
  
def destaque(notas):  
    return sum(notas) / len(notas) >= 9
```

Mudar a fórmula da média obriga a editar todos os pontos.

Hierarquias de Herança Paralelas

Criar uma subclasse exige criar outra em uma hierarquia paralela.



```
class AlunoPresencial extends Aluno {}  
class AlunoEAD extends Aluno {}  
  
class MatriculaPresencial extends Matricula {}  
class MatriculaEAD extends Matricula {}
```

As duas hierarquias crescem juntas e de forma acoplada.

Checkpoint — Qual o cheiro?

```
def aprovado(notas):  
    return sum(notas) / len(notas) >= 7  
  
def bolsista(notas):  
    return sum(notas) / len(notas) >= 8  
  
def destaque(notas):  
    return sum(notas) / len(notas) >= 9
```

Checkpoint — Resposta

Cirurgia de Espingarda: uma única regra (a fórmula da média) está espalhada por vários métodos.

Alterá-la exige lembrar de todos os lugares que a repetem.

Dispensáveis (Dispensables)

Elementos desnecessários que poderiam ser removidos para tornar o sistema mais limpo.

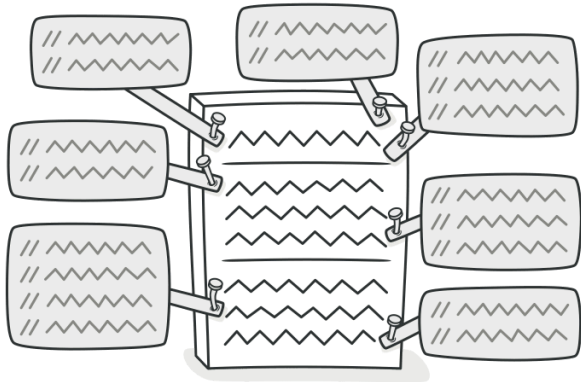


Mapa dos Dispensáveis

Grupo	Cheiro	Sinal de alerta	Onde dói
F	Comentários	comentário explica código confuso	código não se explica
F	Código Duplicado	mesmo trecho repetido	correções inconsistentes
F	Código Morto	ninguém chama	peso e confusão
G	Classe Preguiçosa	faz pouco demais	indireção inútil
G	Classe de Dados	só dados e <i>getters</i>	lógica fora da classe
G	Generalidade Especulativa	criado "para o futuro"	complexidade sem uso

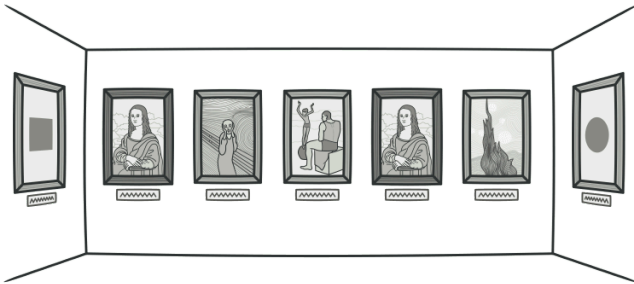
Comentários (*Comments*)

Comentários usados para explicar código confuso.



```
# verifica se o aluno pode se matricular  
# (ativo, sem pendência e dentro do prazo)  
if a.s and not a.p and hoje <= prazo:  
    ...
```

O comentário existe para compensar nomes e condições ruins.



Código Duplicado (*Duplicate Code*)

O mesmo trecho de código aparece em vários lugares.

```
def matricular_presencial(aluno):  
    if aluno.status != "ativo" or aluno.pendencias > 0:  
        raise Erro("Aluno inelegível")  
    ...  
  
def matricular_ead(aluno):  
    if aluno.status != "ativo" or aluno.pendencias > 0:  
        raise Erro("Aluno inelegível")  
    ...
```



Código Morto (*Dead Code*)

Código que nunca é executado ou chamado.

```
def calcular_media_antiga(notas): # ninguém chama desde 2023
    return sum(notas) / len(notas)

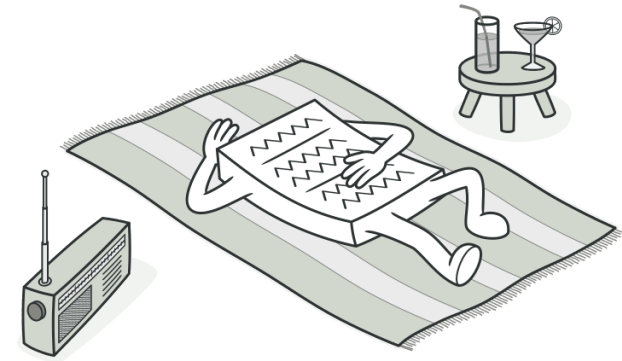
def calcular_media(notas):
    return sum(notas) / len(notas)
```

Classe Preguiçosa (*Lazy Class*)

Classe que não faz o suficiente para justificar sua existência.

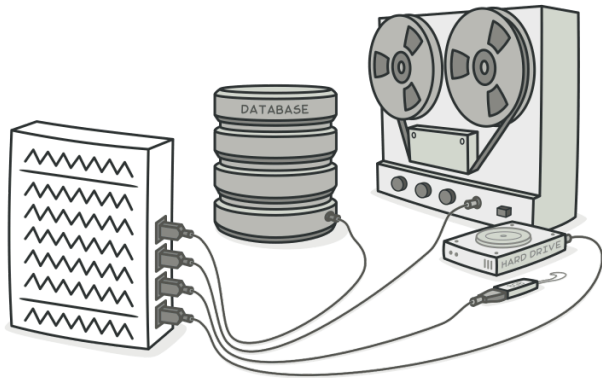
```
class Nota:  
    def __init__(self, valor):  
        self.valor = valor
```

Ela apenas carrega um número, sem agregar comportamento.



Classe de Dados (*Data Class*)

Classe que apenas armazena dados, sem comportamento associado.

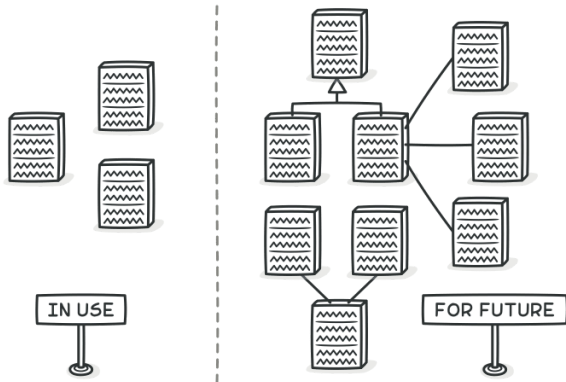


```
class Turma:  
    def __init__(self, codigo, disciplina, vagas):  
        self.codigo = codigo  
        self.disciplina = disciplina  
        self.vagas = vagas
```

Quem quiser saber se há vaga calcula isso **fora** da classe.

Generalidade Especulativa

Código escrito para necessidades futuras que nunca se concretizaram.



```
class Pessoa(ABC):  
    @abstractmethod  
    def get_tipo(self): ...  
  
class Aluno(Pessoa):  
    def get_tipo(self):  
        return "aluno"
```

Abstrações e parâmetros criados "só por garantia".

Checkpoint — Qual o cheiro?

```
# calcula a média ponderada do aluno  
# (prova 1 peso 3, prova 2 peso 3, trabalho peso 4)  
m = (p1 * 3 + p2 * 3 + t * 4) / 10
```

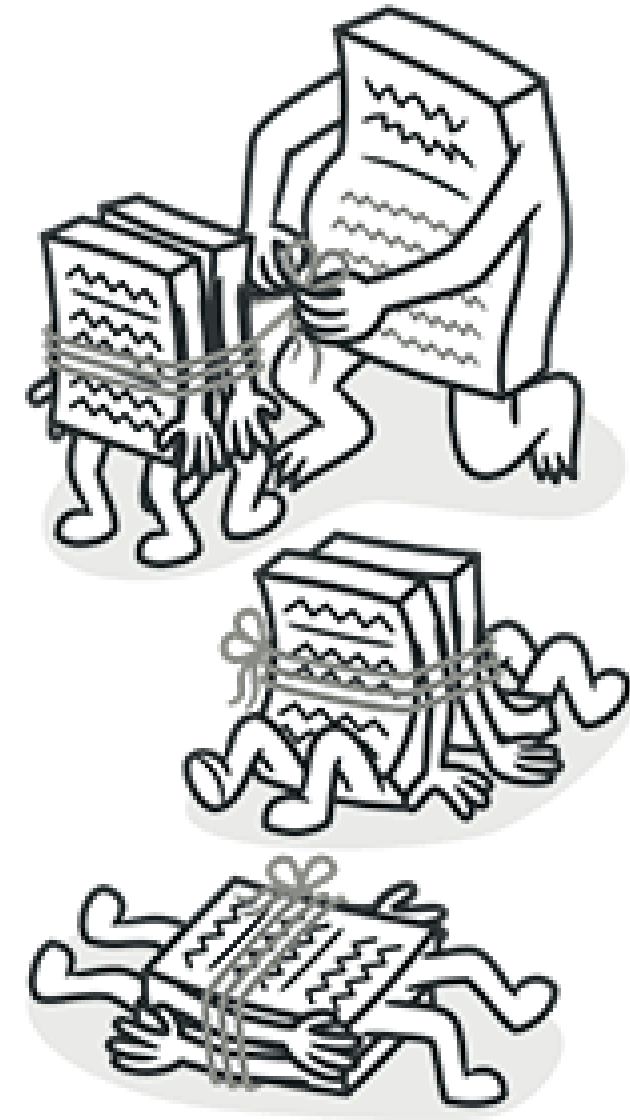
Checkpoint — Resposta

Comentários: o comentário existe para explicar uma expressão que poderia ser **autoexplicativa**.

O sinal é um comentário que descreve **o que** o código faz, e não **por que**.

Acopladores (Couplers)

Problemas relacionados ao acoplamento excessivo entre classes.

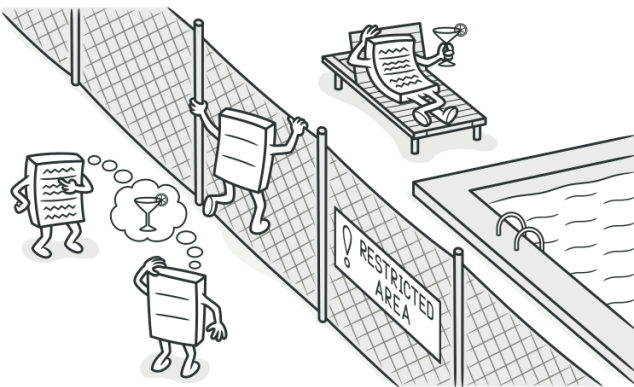


Mapa dos Acopladores

Grupo	Cheiro	Sinal de alerta	Onde dói
H	Inveja de Função	método usa dados de outra classe	lógica no lugar errado
H	Intimidade Inapropriada	classes acessam detalhes internos	acoplamento forte
H	Cadeia de Mensagens	<code>a.getB().getC().getD()</code>	quebra fácil
I	Homem do Meio	só repassa chamadas	indireção inútil
I	Classe de Biblioteca Incompleta	a biblioteca não tem o que preciso	gambiarras locais

Inveja de Função (*Feature Envy*)

Um método acessa dados de outra classe mais do que da própria.

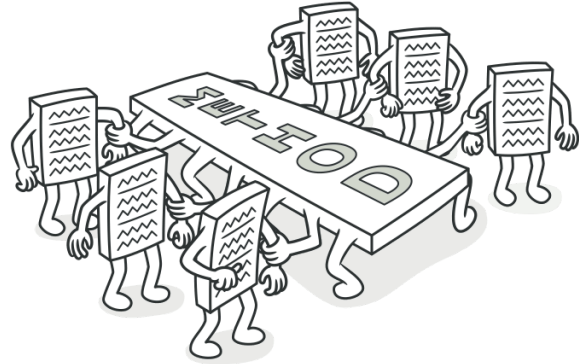


```
class Relatorio:  
    def calcular_media(self, aluno):  
        return sum(aluno.notas) / len(aluno.notas)
```

O método "mora" na classe errada: ele quer os dados do aluno.

Intimidade Inapropriada

Classes que conhecem detalhes internos de outras.



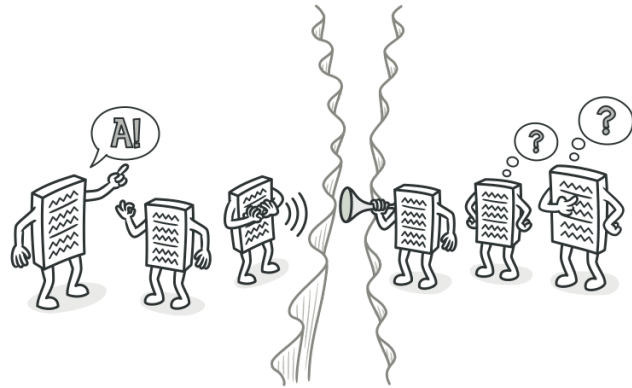
```
class Matricula:
    def __init__(self):
        self._aluno = None

class Secretaria:
    def confirmar(self, matricula):
        matricula._aluno._historico.append("matriculado")
```

A **Secretaria** mexe no interior de objetos que não são dela.

Cadeia de Mensagens (*Message Chains*)

Sequência longa de chamadas encadeadas.

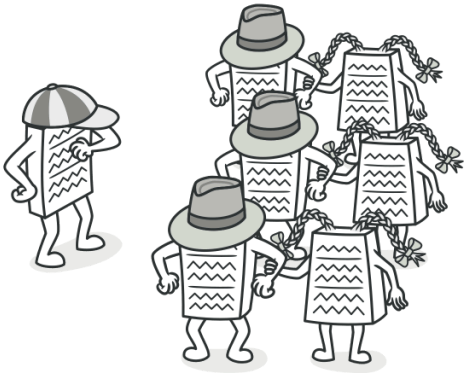


```
def nome_coordenador(aluno):  
    return (aluno.get_matricula()  
            .get_turma()  
            .get_disciplina()  
            .get_curso()  
            .get_coordenador()  
            .get_nome())
```

Qualquer mudança na cadeia quebra quem chama.

Homem do Meio (*Middle Man*)

Classe que apenas repassa chamadas para outros objetos.

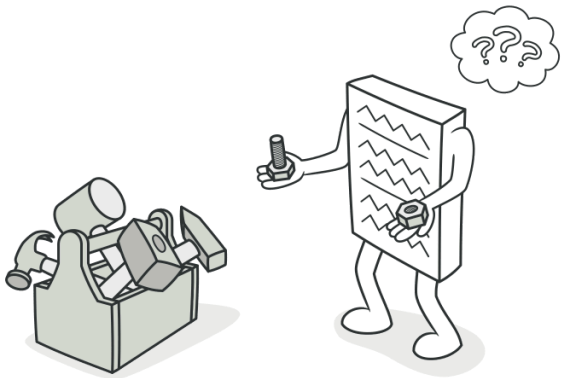


```
class Coordenacao:  
    def __init__(self, secretaria):  
        self.secretaria = secretaria  
  
    def matricular(self, aluno, turma):  
        return self.secretaria.matricular(aluno, turma)
```

A classe não agrega nada: só intermedeia.

Classe de Biblioteca Incompleta

Uma biblioteca de terceiros não oferece o que você precisa.



```
from datetime import date

# a biblioteca não tem "mês letivo"
def mes_letivo(d):
    return d.month if d.day >= 5 else d.month - 1
```

A funcionalidade que falta acaba espalhada em "gambiarras" locais.

Checkpoint — Qual o cheiro?

```
def nome_coordenador(aluno):  
    return (aluno.get_matricula()  
            .get_turma()  
            .get_curso()  
            .get_coordenador()  
            .get_nome())
```

Checkpoint — Resposta

Cadeia de Mensagens: o cliente precisa conhecer toda a estrutura interna da cadeia de objetos.

O sinal é a sequência de `get...().get...().get...()`.

Princípios preventivos

Boa parte desses cheiros surge quando esquecemos princípios já estudados:

- **Nomes claros** evitam comentários explicativos;
- **Funções pequenas** evitam métodos longos;
- **Uma responsabilidade por classe** evita classes grandes e mudanças divergentes;
- **Baixo acoplamento** evita intimidade e cadeias de mensagens;
- **Sem duplicação** evita cirurgias de espingarda.

Prevenir é mais barato do que tratar.

Material de Apoio

- [Refactoring Guru — Code Smells](#)
- [Luzkan — Smells](#)
- [Martin Fowler — Refactoring](#)