TÓPICO 11 - CODE SMELLS

Clean Code - Professor Ramon Venson - SATC 2025

Code Smells

Code smells (ou mal cheiros) são sinais de que algo está errado em seu código.

Um code smell não significa um problema, mas é um alerta de que algo pode ser melhorado.

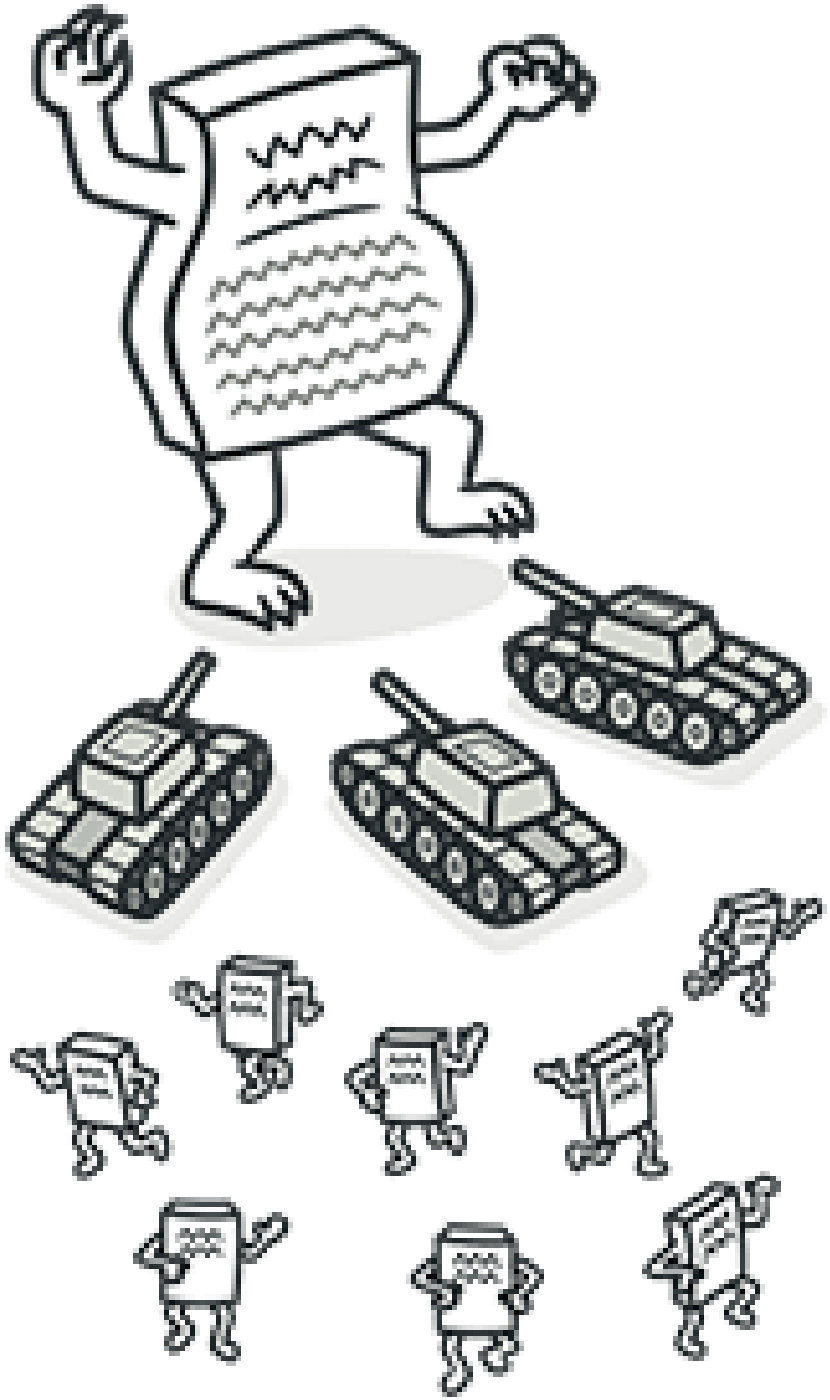
Categorias

Code smells podem ser divididos em categorias:

- Inchaços (Bloaters)
- Abusos da Orientação a Objetos (Object-Orientation Abusers)
- Impedidores de Mudança (Change Preventers)
- Dispensáveis (Dispensables)
- Acopladores (Couplers)

Inchaços (Bloaters)

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



Método Longo (Long Method)

Um método contém linhas demais de código. Geralmente, qualquer método com mais de dez linhas já deveria levantar suspeitas.

Exemplo:

```
def calcula_imposto(usuario):
    leao = Leao()
    renda = usuario.get_pagamentos()
    for pagamento in renda:
        imposto = calcula_imposto_por_renda(pagamento)
        leao.adiciona_imposto(imposto)
    dependentes = usuario.get_dependentes()
    for dependente in dependentes:
        imposto = calcula_imposto_por_dependente(dependente)
        leao.adiciona_imposto(imposto)
    investimentos = usuario.get_investimentos()
    for investimento in investimentos:
        if investimento.is_renda_fixa():
            imposto = calcula_imposto_por_investimento_renda_fixa(investimento)
        else
            imposto = calcula_imposto_por_investimento_renda_variavel(investimento)
        leao.adiciona_imposto(imposto)
```

Como Resolver

Se sentir necessidade de comentar algo dentro de um método, extraia esse trecho em um novo método.

Classe Grande (Large Class)

Uma classe cresceu demais, acumulando responsabilidades diversas.

Exemplo:

```
class Estudante:
    def enviar_email(self, mensagem):
        # lógica para enviar email
    def enviar_sms(self, mensagem):
        # lógica para enviar sms
    def enviar_notificacao(self, mensagem):
        # lógica para enviar notificação
    def alterar_senha(self, nova_senha):
        # lógica para alterar senha
    def registrar_presenca(self, aula):
        # lógica para registrar presença
    def calcular_media(self, notas):
        # lógica para calcular média
```

Como Resolver

Divida a classe em várias classes menores, cada uma focada em uma única responsabilidade.

Obsessão por Primitivos (Primitive Obsession)

Uso excessivo de tipos primitivos (como `int`, `string`) no lugar de objetos específicos.

Exemplo:

```
public class Funcionario {
    private String nome;
    private String cpf;
    private String telefone;
    private String email;
    private String pais;
    private String estado;
    private String cidade;
    private String bairro;
    private String rua;
    private String numero;
    private String complemento;
}
```

Como Resolver

Crie pequenos objetos ou tipos especializados para representar conceitos complexos.

```
public class Endereco {
    private String pais;
    private String estado;
    private String cidade;
    private String bairro;
    private String rua;
    private String numero; // ou seria um int?
    private String complemento;
}
```

Lista Longa de Parâmetros (Long Parameter List)

Método que exige muitos parâmetros, tornando-o difícil de entender e usar.

Exemplo:

```
public String getListaDeProdutos(
    String nome,
    String categoria,
    String marca,
    String tamanho,
    String cor,
    String preco,
    String quantidade) {
    // lógica para buscar produtos
}
```

Como Resolver

Agrupe os parâmetros em objetos ou use o padrão Parameter Object.

```
public String getListaDeProdutos(FiltroProduto filtro) {
    // lógica para buscar produtos
}
```

Agrupamentos de Dados (Data Clumps)

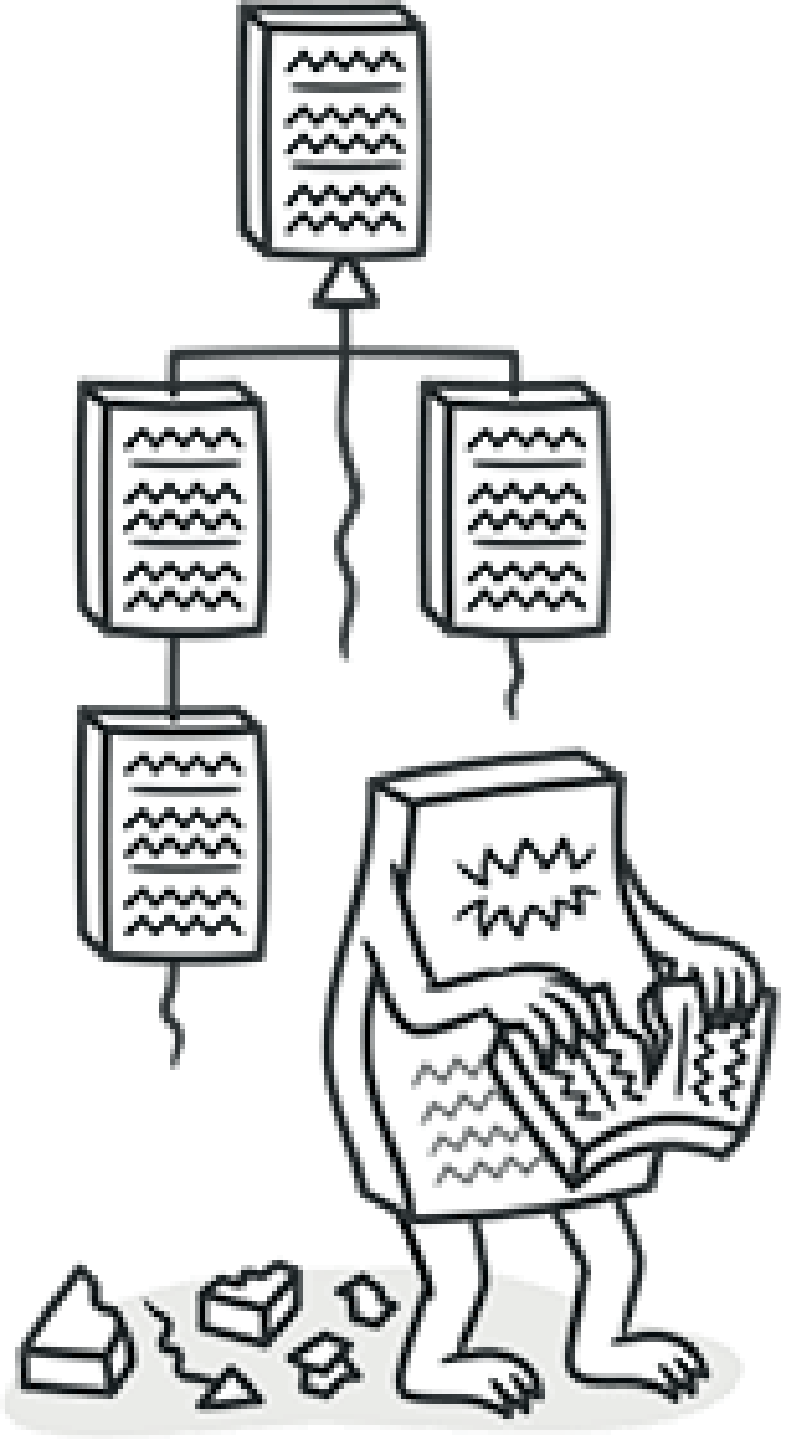
Grupos de variáveis que sempre aparecem juntos.

Exemplo:
````python
def colorize(red: int, green: int, blue: int):
    ...
````

Como Resolver

Agrupe essas variáveis em uma nova classe.

```
class Color:
    def __init__(self, red: int, green: int, blue: int):
        self.red = red
        self.green = green
        self.blue = blue
```



Abusos da Orientação a Objetos (Object-Orientation Abusers)

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

Classes Alternativas com Interfaces Diferentes (Alternative Classes with Different Interfaces)

Classes fazem coisas semelhantes mas têm interfaces diferentes, confundindo quem usa.

Exemplo:

```
class BonecoDeNeve(Humanoide):
    def abraco_boneco_de_neve():
        ...

class Zumbi(Humanoide):
    def abraco_zumbi():
        ...
```

Como Resolver

Unificar as interfaces ou aplicar um padrão de projeto que abstraia as diferenças.

```
class BonecoDeNeve(Humanoide):
    def abraco():
        ...

class Zumbi(Humanoide):
    def abraco():
        ...
```

Recusa de Herança (Refused Bequest)

Uma subclasse herda comportamento de uma superclasse mas não o utiliza.

Exemplo:

```
class Minion(ABC):
    def atacar(self):
        ...

    def mover(self):
        ...

class Tower(Minion):
    def atacar(self):
        ...

    def mover(self):
        raise NotImplementedException
```

Como Resolver

Reorganizar a hierarquia ou preferir composição à herança.

```
class Attackable(ABC):
    def atacar(self):
        ...

class Minion(Attackable):
    def mover(self):
        ...

class Tower(Attackable):
    ...
```

Comandos Switch (Switch Statements)

Uso excessivo de estruturas `switch` ou `if` - `else` para alterar comportamento.

Exemplo:

```
class Exportador:
    def exporte(self, formato: str):
        if formato == 'wav':
            self.exporteEmWav()
        elif formato == 'flac':
            self.exporteEmFlac()
        elif formato == 'mp3':
            self.exporteEmMp3()
        elif formato == 'ogg':
            self.exporteEmOgg()
```

Como Resolver

Substituir por polimorfismo.

```
class Exporter:
    def exporte(self, export_format: str):
        exportador = self.get_format_factory(export_format)
        exportador.exporte()

    def get_format_factory(self, formato: str):
        if formato in self.export_format_factories:
            return render_factory[formato]
        raise MissingFormatException
```

Campo Temporário (Temporary Field)

Campo em um objeto que é usado apenas em situações específicas.

Exemplo:

```
class DateTime:
    def __init__(self, ano, mes, dia):
        self.ano = ano
        self.mes = mes
        self.dia = dia
        self.data_completa = f"{ano}, {mes}, {dia}"

    def foo(self):
        ...

    def bar(self):
        ...

    def __str__(self):
        return self.data_completa
```

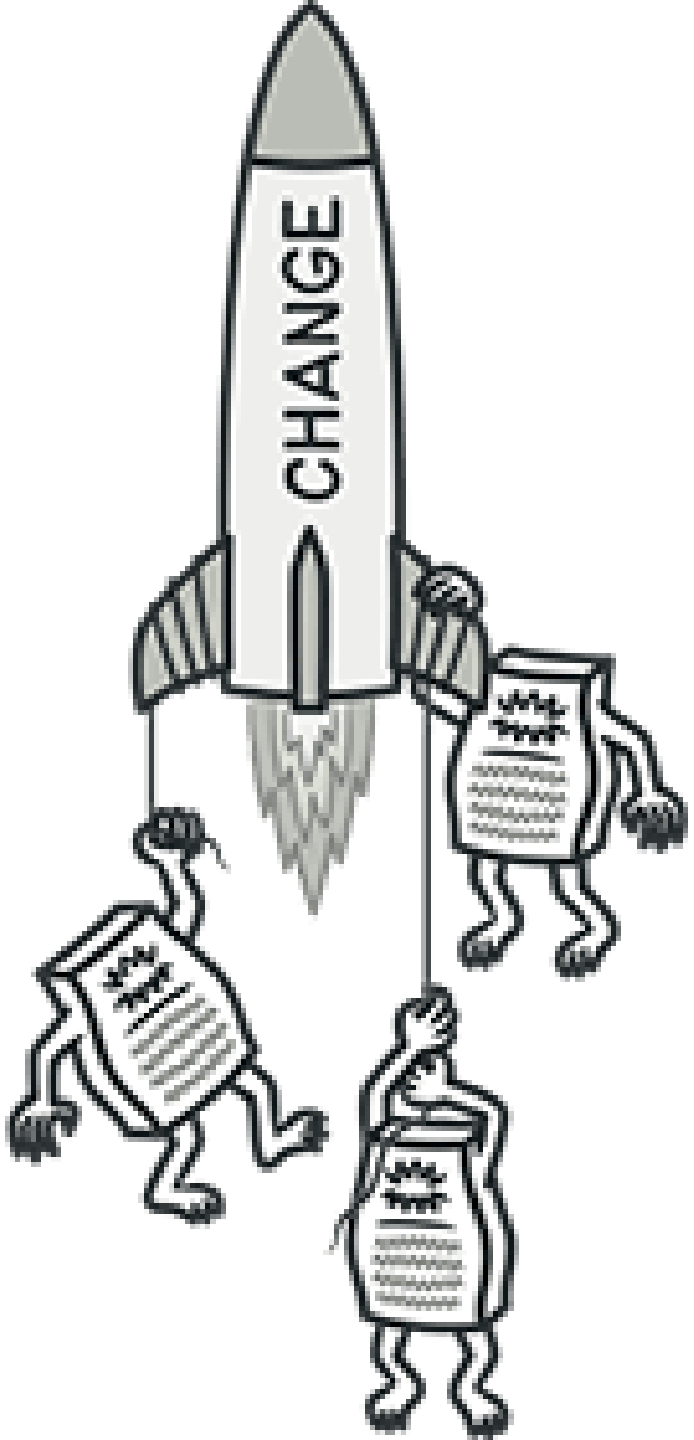
Como Resolver

Extrair uma classe ou método para esses campos temporários.

```
class DateTime:
    def __str__(self):
        return f"{self.ano}, {self.mes}, {self.dia}"
```

Impedidores de Mudança (Change Preventers)

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



Mudança Divergente (Divergent Change)

Uma mesma classe precisa mudar por motivos diferentes.

Exemplo:

```
class AlteradorRelatorio:
    def busca_relatorio(self, nome_relatorio):
        ...
        return relatorio

    def modifica_relatorio(self, relatorio, novo_registro):
        ...
        return relatorio_modificado

    def rodar(self, nome_relatorio, novo_registro):
        relatorio = self.busca_relatorio(nome_relatorio)
        return self.modifica_relatorio(relatorio, novo_registro)

alterador_relatorio = AlteradorRelatorio(...)
relatorio_modificado = alterador_relatorio.rodar('relatorio.csv', 'Nova anotação')
```

Como Resolver

Separar as responsabilidades em diferentes classes.

```
class BuscadorRelatorio:
    def busca_relatorio(self, nome_relatorio):
        ...
        return relatorio

class ModificadorRelatorio:
    def modifica_relatorio(self, relatorio, novo_registro):
        ...
        return relatorio_modificado
```

Hierarquias de Herança Paralelas (Parallel Inheritance Hierarchies)

Sempre que se cria uma subclasse de uma classe, é necessário criar também uma subclasse em outra hierarquia.

Exemplo:

```
class UsuarioBasico() {
    AcessoBasico acesso;
}

class AcessosBasico() {
}

class UsuarioAvancado() {
    AcessoAvancado acesso;
}

class AcessosAvancado() {
}
```

Como Resolver

Fazer com que uma hierarquia delegue comportamento para a outra.

```
class Usuario() {
    Acesso acesso;
}

Usuario = new Usuario();
usuario.acesso = new AcessoBasico();
```

Cirurgia de Espingarda (Shotgun Surgery)

Pequena mudança exige modificações em vários lugares diferentes.

Exemplo:

```
class Minion:
    energy: int

    def attack(self):
        if self.energy < 20:
            animate('no-energy')
            skip_turn()
            return

    def block(self):
        if self.energy < 10:
            animate('no-energy')
            skip_turn()
            return
```

Como Resolver

Agrupar comportamento relacionado em uma única classe ou método.

```
    def has_energy(self, energy_required: int) -> bool:
        if self.energy < energy_required:
            self.handle_no_energy()
            return False
        return True

    def handle_no_energy(self) -> None:
        animate('no-energy')
        skip_turn()
```

Dispensáveis (Dispensables)

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

Comentários (Comments)

Comentários usados para explicar código confuso.

Exemplo:

```
# Criando relatório
vanilla_report = get_vanilla_report(...)
tweaked_report = tweaking_report(vanilla_report)
final_report = format_report(tweaked_report)
# Enviando relatório
send_report_to_headquarters_via_email(final_report)
send_report_to_developers_via_chat(final_report)
```

Como Resolver

Reescrever o código para que ele seja autoexplicativo.

```
def criar_relatorio(self, ...):
    vanilla_report = get_vanilla_report(...)
    tweaked_report = tweaking_report(vanilla_report)
    return format_report(tweaked_report)

def enviar_relatorio(self, report):
    send_report_to_headquarters_via_email(final_report)
    send_report_to_developers_via_chat(final_report)
```

Código Duplicado (Duplicate Code)

Mesmo trecho de código aparece em vários lugares.

Exemplo:

```
function adicionar_h1(texto, parent) {
    var componente = document.createElement('p');
    componente.innerHTML = texto;
    componente.style = 'color: red;';
    componente.onclick = brilhar
    parent.appendChild(componente);
}

function adicionar_h2(texto, parent) {
    var componente = document.createElement('h2');
    componente.innerHTML = texto;
    componente.style = 'color: red;';
    componente.onclick = brilhar
    parent.appendChild(componente);
}
```

Como Resolver

Extrair o código duplicado para um método ou classe comum.

```
function adicionar_titulo(tag, texto, parent) {
    var componente = document.createElement(tag);
    componente.innerHTML = texto;
    componente.style = 'color: red;';
    componente.onclick = brilhar
    parent.appendChild(componente);
}
```

Classe de Dados (Data Class)

Classe que apenas armazena dados sem comportamento associado.

Exemplo:

```
class Navegador:
    def __init__(self, url):
        self.url = url
        self.historico = []
```

Como Resolver

Adicionar comportamentos relevantes à classe.

```
class Navegador:
    def navegar(self, url):
        self.url = url
        self.historico.append(url)

    def limpar_historico(self):
        self.historico = []
```

Código Morto (Dead Code)

Código que nunca é utilizado.

Exemplo:

```
def calcular_desconto(renda):
    return renda * 0.1

def calcular_imposto(renda):
    if renda < 1000:
        return renda * 0.1
    else:
        return renda * 0.2

calcular_imposto(1000)
```

Como Resolver

Remover o código não utilizado.

Classe Preguiçosa (Lazy Class)

Classe que não faz o suficiente para justificar sua existência.

Exemplo:

```
class Forca:
    valor: int

class Personagem:
    vida: int
    inteligencia: int
    forca: Forca
```

Como Resolver

Incorporar sua funcionalidade em outra classe.

```
class Personagem:
    vida: int
    inteligencia: int
    forca: int
```

Generalidade Especulativa (Speculative Generality)

Código escrito para atender a necessidades futuras que nunca se concretizaram.

Exemplo:

```
class Animal:
    vida: int

class Humano(Animal):
    nome: str
    ataque: int
    defesa: int

class Guerreiro(Humano):
    ...

class Arqueiro(Humano):
    ...

class Mago(Humano):
    ...
```

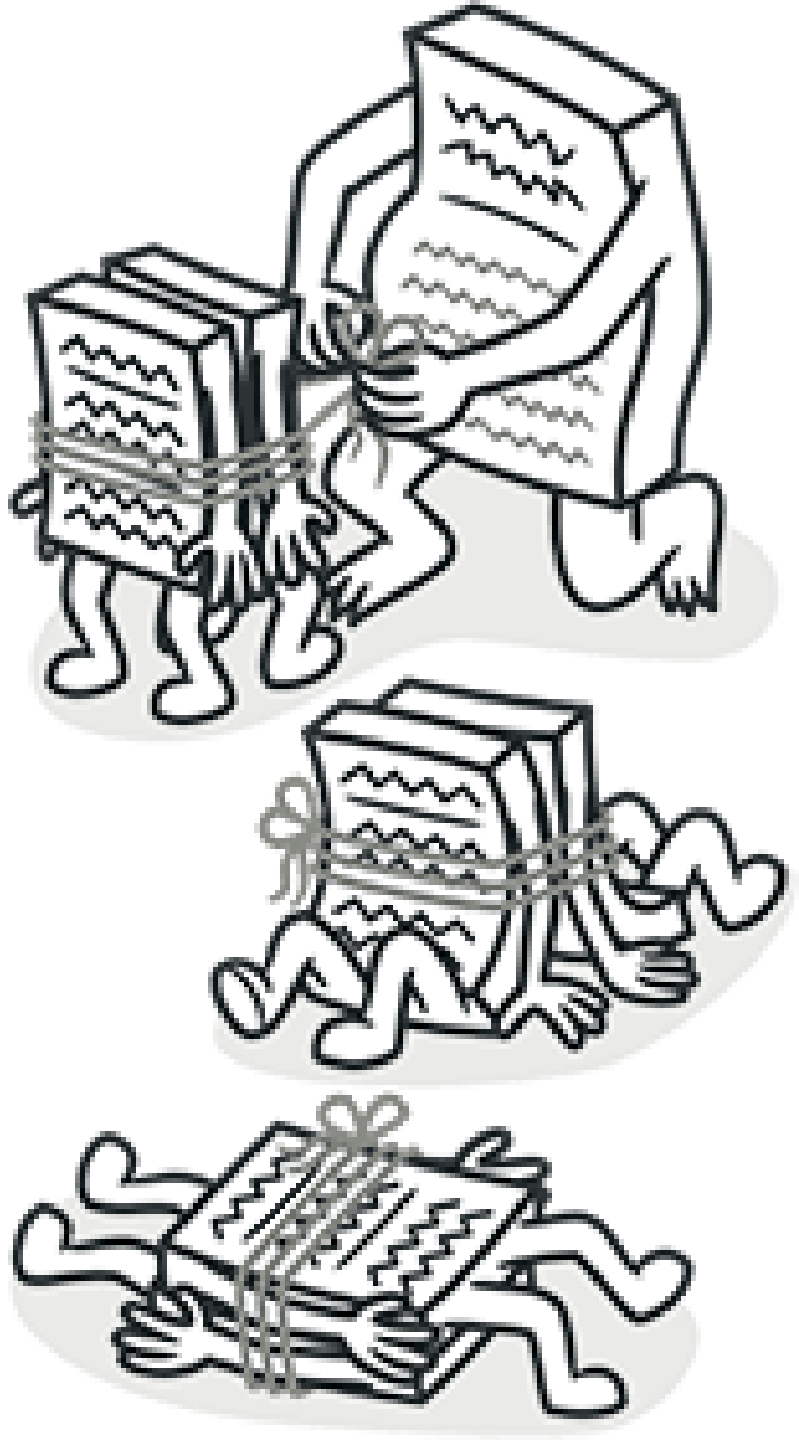
Como Resolver

Remover ou simplificar o código.

```
class Humano:
    nome: str
    ataque: int
    defesa: int
    vida: int
```

Acopladores (Couplers)

Problemas relacionados ao acoplamento excessivo entre classes.



Inveja de Função (Feature Envy)

Método que acessa dados de outra classe mais do que de sua própria classe.

Exemplo:

```
class Pedido:
    def calcular_total(self, itens: list[ItemCompra]) -> float:
        return sum([item.preco * item.imposto for item in itens])

    def gerar_lista_recibo(self, itens: list[ItemCompra]) -> list[str]:
        return [f"{item.nome}: {item.preco * item.imposto}R$" for item in itens]

    def criar_recibo(self, itens: list[ItemCompra]) -> str:
        total = self.calcular_total(itens)
        recibo = '\n'.join(self.gerar_lista_recibo(itens))
        return f"{recibo}\nTotal {total}R$"
```

Como Resolver

Mover o método para a classe onde está o dado que ele usa.

```
class ItemCompra:
    nome: str
    preco: float
    imposto: float

    def preco_com_imposto(self) -> float:
        return self.preco * self.imposto

    def gerar_linha_recibo(self) -> str:
        return f"{self.nome}: {self.preco_com_imposto}R$"
```

Intimidade Inapropriada (Inappropriate Intimacy)

Classes que conhecem detalhes privados de outras.

Exemplo:

```
class Contador {
    public int contagem = 0;

    public void incrementar() {
        contagem++;
    }
}

contador = new Contador();
contador.incrementar();
System.out.println(contador.contagem);
```

Como Resolver

Reduzir o acoplamento entre essas classes.

```
class Contador {
    private int contagem = 0;

    public void incrementar() {
        contagem++;
    }

    public int getContagem() {
        return contagem;
    }
}

contador = new Contador();
contador.incrementar();
System.out.println(contador.getContagem());
```

Classe de Biblioteca Incompleta (Incomplete Library Class)

Classe de biblioteca que não oferece tudo o que é necessário.

Exemplo:

```
var original = File()
if original.file_exists("user://dados.txt"):
    original.open("user://dados.txt", File.READ)
    var content = original.get_as_text()
    original.close()

    var copia = File.new()
    copia.open("user://dados_copia.txt", File.WRITE)
    copia.store_string(content)
    copia.close()

    # Agora, se quiser "mover", teria que excluir o arquivo antigo manualmente:
    DirAccess.remove("user://dados.txt")
```

Como Resolver

Estender a classe ou criar wrappers que adicionem a funcionalidade.

```
class FileManager:
    def move(self, file, destino):
        file.copy(destino)
        file.delete()
```

Cadeia de Mensagens (Message Chains)

Sequência longa de chamadas de método (ex: `a.getB().getC().getD()`).

Exemplo:

```
def calcula_imposto_devido(usuario, imposto_pago):
    imposto_total = usuario.get_declaracao().get_renda().get_imposto()
    return imposto_total - imposto_pago
```

Como Resolver

Criar métodos intermediários para esconder a cadeia de chamadas.

```
def calcula_imposto_devido(usuario, imposto_pago):
    imposto_total = usuario.get_total_imposto()
    return imposto_total - imposto_pago
```

Homem do Meio (Middle Man)

Classe que apenas repassa chamadas para outros objetos.

Exemplo:

```
def realiza_requisicao():
    resposta = requisicao()
    dados = recebe_resposta(resposta)
    return dados

def recebe_resposta(resposta):
    return resposta.get_dados()
```

Como Resolver

Eliminar o intermediário e deixar o cliente lidar diretamente com o objeto real.

```
def realiza_requisicao():
    resposta = requisicao()
    return resposta.get_dados()
```

Material de Apoio

- Refactoring Guru
- Luzkan