2. Object-Orientation Abusers (Abusadores de Orientação a Objetos)

Esta categoria de *code smells* surge quando os princípios fundamentais da programação orientada a objetos (POO) – como encapsulamento, herança, polimorfismo e abstração – são aplicados de forma incompleta, incorreta ou simplesmente ignorados. Tais abusos podem levar a um design de software menos flexível, mais difícil de estender e manter, e frequentemente mais complexo do que o necessário. Problemas comuns incluem hierarquias de classes confusas, responsabilidades mal distribuídas e código que resiste a mudanças de forma elegante. O impacto geral é um sistema que não colhe todos os benefícios prometidos pelo paradigma OO, como reutilização, manutenibilidade e extensibilidade.

2.1. Switch Statements (Instruções Switch / Condicionais Complexas)

- **Definição:** Este *smell* ocorre com o uso excessivo de instruções switch (ou cadeias longas e complexas de if-else-if) que tomam decisões baseadas no tipo de um objeto ou em algum atributo que funciona como um código de tipo, para então determinar o comportamento a ser executado. Em programação orientada a objetos, o uso de switch para simular comportamento polimórfico é frequentemente considerado um anti-padrão.
- **Sintomas e Causas:** Uma característica comum é que a lógica de um único switch pode estar replicada ou espalhada por diferentes partes do sistema. Consequentemente, adicionar uma nova condição (um novo "tipo" ou "caso") exige encontrar e modificar todas as ocorrências dessas estruturas switch. Isso viola diretamente o Princípio Aberto/Fechado (OCP), que preconiza que entidades de software devem ser abertas para extensão, mas fechadas para modificação. O *smell* é particularmente forte quando o switch opera sobre um "código de tipo" que define o comportamento de um objeto; nesses casos, o polimorfismo é a solução idiomática em POO.
  É importante notar que, em linguagens como Python 3.10+, a introdução da instrução match-case oferece uma forma mais estruturada e poderosa de lidar com múltiplos condicionais em comparação com longas sequências de if-elif-else. No entanto, mesmo com match-case, se o objetivo é selecionar comportamento com base no tipo de um objeto, o polimorfismo ainda é geralmente a abordagem de design OO preferida. O LLM deve ser capaz de distinguir uma melhoria sintática (como match-case sobre if-elif) de uma melhoria de design OO (polimorfismo sobre switch em tipo).

- **Exemplos de Código:**
  - Java:
    O exemplo apresenta uma classe Bird com um método getSpeed() que usa switch (type) para determinar a velocidade com base no tipo da ave.
    Antes da Refatoração:

```
Java
enum BirdType { EUROPEAN, AFRICAN, NORWEGIAN_BLUE }

class Bird {
   private BirdType type;
   private double baseSpeed;
   private double loadFactor;
   private int numberOfCoconuts;
   private boolean isNailed;
   private double voltage; // Apenas para Norwegian Blue

   // Construtor e outros métodos omitidos
   public Bird(BirdType type, double baseSpeed, double loadFactor, int numberOfCoconuts,
boolean isNailed, double voltage) {
      this.type = type;
      this.baseSpeed = baseSpeed;
      this.loadFactor = loadFactor;
      this.numberOfCoconuts = numberOfCoconuts;
      this.isNailed = isNailed;
      this.voltage = voltage;
   }

   public double getBaseSpeed() { return baseSpeed; }
   public double getLoadFactor() { return loadFactor; }
   public int getNumberOfCoconuts() { return numberOfCoconuts; }
   public boolean isNailed() { return isNailed; }
   public double getBaseSpeed(double voltage) { return baseSpeed - voltage; } //
Exemplo para Norwegian Blue

   double getSpeed() {
      switch (type) {
         case EUROPEAN:
```

```
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * getNumberOfCoconuts();
        case NORWEGIAN_BLUE:
            return (isNailed())? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

Após a Refatoração (Usando Polimorfismo):

```
abstract class AbstractBird {
  protected double baseSpeed;
  protected double loadFactor;
  protected int numberOfCoconuts;
  protected boolean isNailed;
  protected double voltage;

  public AbstractBird(double baseSpeed, double loadFactor, int numberOfCoconuts,
boolean isNailed, double voltage) {
    this.baseSpeed = baseSpeed;
    this.loadFactor = loadFactor;
    this.numberOfCoconuts = numberOfCoconuts;
    this.isNailed = isNailed;
    this.voltage = voltage;
  }

  public double getBaseSpeed() { return baseSpeed; }
  public double getLoadFactor() { return loadFactor; }
  public int getNumberOfCoconuts() { return numberOfCoconuts; }
  public boolean isNailed() { return isNailed; }
  public double getBaseSpeed(double appliedVoltage) { return baseSpeed -
appliedVoltage; }

  abstract double getSpeed();
}
```

```
class EuropeanBird extends AbstractBird {
    public EuropeanBird(double baseSpeed) { super(baseSpeed, 0,0,false,0); }
    @Override
    double getSpeed() {
        return getBaseSpeed();
    }
}

class AfricanBird extends AbstractBird {
     public AfricanBird(double baseSpeed, double loadFactor, int coconuts) {
super(baseSpeed, loadFactor,coconuts,false,0); }
    @Override
    double getSpeed() {
        return getBaseSpeed() - getLoadFactor() * getNumberOfCoconuts();
    }
}

class NorwegianBlueBird extends AbstractBird {
    public NorwegianBlueBird(double baseSpeed, boolean isNailed, double voltage) {
super(baseSpeed, 0,0,isNailed,voltage); }
    @Override
    double getSpeed() {
        return (isNailed())? 0 : getBaseSpeed(this.voltage);
    }
}
// Uso:
// AbstractBird bird = new AfricanBird(10.0, 0.5, 2);
// double speed = bird.getSpeed();
```

- **Técnicas de Refatoração Sugeridas:** A solução mais idiomática em POO é **Substituir Condicional por Polimorfismo** (*Replace Conditional with Polymorphism*).[13] Isso geralmente envolve a criação de uma superclasse ou interface que define um método abstrato, e subclasses que fornecem implementações concretas desse método, correspondendo a cada caso do switch. Padrões de projeto como **Strategy** [13] ou **State** também são aplicáveis. Para casos mais simples, onde o switch apenas seleciona um valor ou uma ação simples, um mapeamento para um dicionário (ou Map em Java/C#) de funções ou ações pode ser uma alternativa eficaz. Outras técnicas incluem **Extrair Método**

(*Extract Method*) e **Mover Método** (*Move Method*) para primeiro isolar a lógica do switch na classe apropriada antes de aplicar o polimorfismo.
- **Impacto da Refatoração:** O código torna-se mais alinhado com os princípios da POO, especialmente o Princípio Aberto/Fechado. Adicionar novos comportamentos (novos "casos") geralmente envolve adicionar uma nova subclasse/estratégia, sem modificar o código existente que usa a abstração. Isso torna o sistema mais fácil de estender e manter.

2.2. Temporary Field (Campo Temporário)

- **Definição:** Um *Temporary Field* é um atributo (campo) de uma classe que só recebe um valor e é efetivamente utilizado sob certas circunstâncias ou durante a execução de um algoritmo específico. Fora desses contextos, o campo permanece vazio, nulo ou com um valor indefinido.[15]
- **Sintomas e Causas:** A causa mais comum para a criação de campos temporários é a tentativa de evitar listas de parâmetros muito longas em um método que implementa um algoritmo complexo. Em vez de passar múltiplos valores como parâmetros, o programador opta por armazená-los como campos da classe. Esses campos são, então, usados exclusivamente por esse algoritmo e permanecem ociosos (e muitas vezes sem inicialização válida) durante o resto do ciclo de vida do objeto. Isso torna o código difícil de entender, pois os desenvolvedores esperam que os campos de um objeto contenham dados relevantes para o estado do objeto de forma consistente, mas encontram campos que estão quase sempre vazios ou sem sentido.
  É importante distinguir o *code smell* "Temporary Field" (um atributo de classe) de uma "variável temporária" (uma variável local a um método). O *smell* "Temporary Field" refere-se ao atributo de classe que tem seu valor definido apenas condicionalmente. A refatoração "Split Temporary Variable" aplica-se ao caso de variáveis locais reutilizadas para múltiplos propósitos dentro de um método, o que é mais um problema de clareza do método do que um abuso dos princípios de OO. O campo temporário de classe é mais problemático para a coesão e o gerenciamento do estado do objeto. Frequentemente, os campos temporários são um indicativo de que um método cresceu demais ou está tentando gerenciar um estado que deveria ser encapsulado em um objeto de método ou passado explicitamente.
- **Exemplos de Código:**
  - Java:
    O exemplo descreve uma classe Estimator que possui campos como

durations, average e standardDeviation. Esses campos são inicializados e usados apenas dentro do método CalculateEstimate para evitar passar múltiplos parâmetros.

Antes da Refatoração (Campos temporários em Estimator):

```Java
import java.util.List;
import java.time.Duration;

public class Estimator {
    private final Duration defaultEstimate;
    // Campos temporários para o método CalculateEstimate
    private List<Duration> durations;
    private Duration average;
    private Duration standardDeviation;

    public Estimator(Duration defaultEstimate) {
        this.defaultEstimate = defaultEstimate;
    }

    private void calculateStats() {
        if (this.durations == null || this.durations.isEmpty()) {
this.average = defaultEstimate; // Ou alguma outra lógica
this.standardDeviation = Duration.ZERO;
return;
}
// Lógica para calcular média (average) e desvio padrão (standardDeviation)
// a partir de this.durations
long sum = 0;
for (Duration d : this.durations) sum += d.toMillis();
this.average = Duration.ofMillis(sum / this.durations.size());
//... cálculo do desvio padrão...
this.standardDeviation = Duration.ofMinutes(5); // Valor fictício
}

    public Duration CalculateEstimate(List<Duration> observedDurations) {
        this.durations = observedDurations; // Campo temporário recebe valor
        calculateStats(); // Usa os campos temporários
```

```
        if (this.durations == null |

| this.durations.isEmpty()) {
return this.defaultEstimate;
}
// Lógica complexa usando this.average e this.standardDeviation
return this.average.plus(this.standardDeviation.multipliedBy(2)); // Exemplo
}
}
```

Após a Refatoração (Extraindo uma classe `DurationStatistics` - Objeto de Método):

```java
import java.util.List;
import java.time.Duration;

class DurationStatistics { // Nova classe para encapsular os "campos temporários" e sua lógica
    private final List<Duration> durations;
    private final Duration average;
    private final Duration standardDeviation;
    private final Duration defaultEstimate;

    public DurationStatistics(List<Duration> durations, Duration defaultEstimate) {
        this.durations = durations;
        this.defaultEstimate = defaultEstimate;

        if (this.durations == null || this.durations.isEmpty()) {
this.average = defaultEstimate;
this.standardDeviation = Duration.ZERO;
} else {
long sum = 0;
for (Duration d : this.durations) sum += d.toMillis();
this.average = Duration.ofMillis(sum / this.durations.size());
//... cálculo do desvio padrão...
this.standardDeviation = Duration.ofMinutes(5); // Valor fictício
}
}
    public Duration getEstimate() {
```

```
        if (this.durations == null |

| this.durations.isEmpty()) {
return this.defaultEstimate;
}
return this.average.plus(this.standardDeviation.multipliedBy(2));
}
}

  public class EstimatorRefactored {
    private final Duration defaultEstimate;

    public EstimatorRefactored(Duration defaultEstimate) {
      this.defaultEstimate = defaultEstimate;
    }

    public Duration CalculateEstimate(List<Duration> observedDurations) {
      DurationStatistics stats = new DurationStatistics(observedDurations,
this.defaultEstimate);
      return stats.getEstimate();
    }
  }
```

- **Técnicas de Refatoração Sugeridas:**
  - **Extrair Classe** (*Extract Class*): Os campos temporários e todo o código que opera sobre eles podem ser movidos para uma nova classe separada. Isso é efetivamente a criação de um **Objeto de Método** (*Method Object*), que encapsula o algoritmo e seus dados temporários.[15]
  - **Introduzir Objeto Nulo** (*Introduce Null Object*): Se o código cliente frequentemente verifica se os campos temporários têm valores válidos (não nulos), um Objeto Nulo pode ser usado para substituir essas verificações condicionais, simplificando o código cliente e fornecendo um comportamento padrão para o caso "vazio".[15]
- **Impacto da Refatoração:** A principal vantagem é uma melhora na clareza e organização do código.[15] O estado do objeto original torna-se mais consistente e previsível, pois não é mais poluído por campos que são relevantes apenas esporadicamente. A lógica do algoritmo complexo fica encapsulada em seu próprio objeto, facilitando o entendimento e a manutenção.

2.3. Refused Bequest (Herança Recusada)

- **Definição:** Este *smell* ocorre quando uma subclasse herda métodos e propriedades de sua superclasse, mas deliberadamente não os utiliza, os redefine para não fazer nada (no-op), ou os redefine de uma maneira que viola o contrato ou a intenção da superclasse.[16] Isso indica que a hierarquia de herança pode estar mal concebida ou que a relação "é um tipo de" não se sustenta completamente.
- **Sintomas e Causas:** A principal causa é, frequentemente, uma motivação equivocada para usar herança: o desejo de reutilizar código de uma superclasse, mesmo quando a relação conceitual entre superclasse e subclasse é fraca ou inexistente. A subclasse pode sobrescrever métodos herdados para lançar exceções como UnsupportedOperationException ou simplesmente não fornecer uma implementação útil. Este *smell* é uma violação do Princípio de Substituição de Liskov (LSP), que afirma que instâncias de subclasses devem ser substituíveis por instâncias de sua superclasse sem alterar a corretude do programa.
  É crucial distinguir entre uma sobrescrita polimórfica válida (onde a subclasse fornece uma implementação especializada que ainda honra o contrato da superclasse) e uma "recusa" da herança. Recusar a *implementação* (sobrescrevendo um método com comportamento diferente, mas compatível) é uma parte normal e esperada da POO. O cerne do *smell* "Refused Bequest" está em recusar a *interface* ou o contrato da superclasse. O desejo de reuso de código é uma causa frequente de herança inadequada que leva a este *smell*; muitas vezes, a composição sobre a herança seria uma solução de design mais apropriada e robusta.
- **Exemplos de Código:**
  - Java:
    O exemplo em e descreve uma classe Plane que herda de Vehicle. Se Vehicle tem um método Drive(), isso não faria sentido para Plane. Outro exemplo é uma classe Person que herda de Tax, mas o método GetTaxAmount() da subclasse Person ignora completamente a implementação da superclasse Tax e fornece uma lógica totalmente nova, "recusando" a implementação herdada.
    Antes da Refatoração (Exemplo Vehicle/Plane conceitual):

```
Java
class Vehicle {
    public void StartEngine() { System.out.println("Engine started."); }
    public void StopEngine() { System.out.println("Engine stopped."); }
```

```
    public void Drive() { System.out.println("Vehicle is driving."); } // Relevante para Carro, não para Avião
}

class Car extends Vehicle {
    @Override
    public void Drive() { System.out.println("Car is driving on road."); }
}

class Plane extends Vehicle {
    @Override
    public void Drive() {
        // Um avião não "dirige" no mesmo sentido.
        // Poderia lançar UnsupportedOperationException ou não fazer nada.
        // Ou, pior, implementar algo totalmente diferente que não é "dirigir".
        System.out.println("Plane cannot 'drive' in the conventional sense.");
        // throw new UnsupportedOperationException("Planes fly, they don't drive.");
    }

    public void Fly() { System.out.println("Plane is flying."); }
}
```

Após a Refatoração (Empurrar Método para Baixo ou Usar Interfaces/Composição):
Uma solução seria não ter Drive() em Vehicle ou torná-lo abstrato e apenas implementá-lo onde faz sentido, ou usar interfaces separadas como IDrivable, IFlyable.

```
Java
// Solução 1: Empurrar para baixo / Remover da base
class VehicleRefactored { // Vehicle base mais genérico
    public void StartEngine() { System.out.println("Engine started."); }
    public void StopEngine() { System.out.println("Engine stopped."); }
}

class CarRefactored extends VehicleRefactored {
    public void Drive() { System.out.println("Car is driving on road."); }
}
```

```
class PlaneRefactored extends VehicleRefactored {
    public void Fly() { System.out.println("Plane is flying."); }
}

// Solução 2: Usar interfaces (mais flexível)
interface IEnginePowered {
    void StartEngine();
    void StopEngine();
}
interface IDrivable extends IEnginePowered {
    void Drive();
}
interface IFlyable extends IEnginePowered {
    void Fly();
}

class CarWithInterface implements IDrivable {
    @Override public void StartEngine() { System.out.println("Car engine started."); }
    @Override public void StopEngine() { System.out.println("Car engine stopped."); }
    @Override public void Drive() { System.out.println("Car is driving."); }
}
class PlaneWithInterface implements IFlyable {
    @Override public void StartEngine() { System.out.println("Plane engine started."); }
    @Override public void StopEngine() { System.out.println("Plane engine stopped."); }
    @Override public void Fly() { System.out.println("Plane is flying."); }
}
```

- **Técnicas de Refatoração Sugeridas:**
  - **Substituir Herança por Delegação** (*Replace Inheritance with Delegation*): Se a relação "é um tipo de" não se sustenta e a herança foi usada primariamente para reuso de código, a subclasse pode conter uma instância da (antiga) superclasse e delegar chamadas aos métodos relevantes, expondo apenas a interface desejada.
  - **Empurrar Método para Baixo / Empurrar Campo para Baixo** (*Push Down Method / Push Down Field*): Se a herança é conceitualmente apropriada, mas a superclasse é muito geral, funcionalidades (métodos ou campos) que não são usadas pela maioria das subclasses devem ser movidas da superclasse

para as subclasses específicas que realmente as utilizam.
    - **Extrair Superclasse** (*Extract Superclass*): Se diferentes subconjuntos de subclasses usam diferentes partes da superclasse, pode ser útil extrair uma ou mais superclasses intermediárias que contenham apenas o comportamento compartilhado relevante para cada grupo de subclasses.
- **Impacto da Refatoração:** Resulta em hierarquias de classes mais claras, lógicas e coesas. Melhora a adesão ao Princípio de Substituição de Liskov, levando a um design mais robusto e menos propenso a erros quando se trabalha com polimorfismo.

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

- **Definição:** Este *smell* ocorre quando duas ou mais classes desempenham funções idênticas ou muito semelhantes, mas o fazem através de métodos com nomes diferentes ou com assinaturas (parâmetros, tipo de retorno) distintas.[17]
- **Sintomas e Causas:** A causa mais provável é a falta de comunicação ou conhecimento: um desenvolvedor cria uma classe para uma determinada funcionalidade sem saber que uma classe com propósito similar já existe no sistema, possivelmente criada por outro desenvolvedor ou em outra parte do projeto. Isso leva à duplicação de funcionalidade sob diferentes "disfarces" (interfaces). A falta de uma interface comum abstrata ou de convenções de nomenclatura também contribui para este *smell*. Este *smell* destaca a importância de um bom design de interface e de comunicação eficaz dentro da equipe de desenvolvimento.
  Resolver este *smell* pode ser particularmente desafiador em código legado ou ao integrar bibliotecas de terceiros, onde modificar as interfaces das classes alternativas pode não ser viável. Nesses casos, o padrão Adapter torna-se uma ferramenta de refatoração crucial.
- **Exemplos de Código:**
    - Java:

```
// Exemplo conceitual baseado em [18], S77
public class Person { /*... */ }

public class BinaryParser // Não implementa interface comum
{
    public void Open(string filePath) { /* Lógica para abrir arquivo binário */ }
    public bool HasReachedEnd { get; /* Lógica para verificar fim do arquivo */ }
```

```
    public Person GetPerson() { /* Lógica para ler pessoa do binário */ return new Person();
}
    public void Close() { /* Lógica para fechar arquivo binário */ }
}

public class XmlParser // Não implementa interface comum
{
    public XmlParser(string filePath) { /* Lógica para abrir arquivo XML no construtor */ }
    public void StartParse() { /* Preparação para parsear XML */ }
    public Person GetNextPerson() { /* Lógica para ler pessoa do XML, retorna null no fim */
return new Person(); }
    public void FinishParse() { /* Finalização do parse XML */ }
}

// Código cliente usando os parsers:
public List<Person> LoadPersons(string filePath, string sourceType)
{
    var persons = new List<Person>();
    if (sourceType == "bin")
    {
        var parser = new BinaryParser();
        parser.Open(filePath);
        while (!parser.HasReachedEnd)
        {
            persons.Add(parser.GetPerson());
        }
        parser.Close();
    }
    else if (sourceType == "xml")
    {
        var parser = new XmlParser(filePath);
        parser.StartParse();
        Person p;
        while ((p = parser.GetNextPerson())!= null)
        {
            persons.Add(p);
        }
```

```
        parser.FinishParse();
    }
    return persons;
}
```

Após a Refatoração (Introduzindo interface IParser e ParserFactory):

Java

```
public interface IParser : IDisposable
{
    Person GetNext(); // Método unificado para obter a próxima pessoa
}

public class BinaryParserRefactored : IParser
{
    // Construtor agora abre o arquivo
    public BinaryParserRefactored(string filePath) { /*... */ }
    public Person GetNext() { /* Lógica adaptada... retorna null no fim */ return new Person();
}
    public void Dispose() { /* Fecha o arquivo */ }
}

public class XmlParserRefactored : IParser
{
    // Construtor agora abre e inicia o parse
    public XmlParserRefactored(string filePath) { /*... */ }
    public Person GetNext() { /* Lógica adaptada... retorna null no fim */ return new Person();
}
    public void Dispose() { /* Fecha o leitor XML */ }
}

public static class ParserFactory
{
    public static IParser Create(string filePath, string sourceType)
    {
        switch (sourceType.ToLower())
        {
            case "bin": return new BinaryParserRefactored(filePath);
```

```
            case "xml": return new XmlParserRefactored(filePath);
            default: throw new NotSupportedException($"Source type '{sourceType}' not
supported.");
        }
    }
}

// Código cliente refatorado:
public List<Person> LoadPersonsRefactored(string filePath, string sourceType)
{
    var persons = new List<Person>();
    using (IParser parser = ParserFactory.Create(filePath, sourceType))
    {
        Person p;
        while ((p = parser.GetNext())!= null)
        {
            persons.Add(p);
        }
    }
    return persons;
}
```

- **Técnicas de Refatoração Sugeridas:**
  - **Renomear Métodos** (*Rename Method*): Se as funcionalidades são idênticas, renomear os métodos para que tenham o mesmo nome e assinatura é o primeiro passo.
  - **Mover Método** (*Move Method*), **Adicionar Parâmetro** (*Add Parameter*), **Parametrizar Método** (*Parameterize Method*): Estas técnicas podem ser usadas para ajustar as assinaturas dos métodos e suas implementações para que se tornem verdadeiramente idênticas ou compatíveis com uma interface comum.
  - **Extrair Superclasse** (*Extract Superclass*): Se as classes compartilham apenas parte de sua funcionalidade, pode-se extrair a parte comum para uma superclasse, da qual as classes existentes herdarão. A interface comum pode então ser definida na superclasse.
  - **Unificar Interfaces com Adapter** (*Unify Interfaces with Adapter*): Se não for possível modificar as classes existentes (e.g., são de bibliotecas de terceiros),

o padrão de projeto Adapter pode ser usado para criar um wrapper em torno de uma ou ambas as classes, expondo uma interface unificada.
    - Após aplicar essas técnicas, pode ser possível eliminar uma das classes alternativas se ela se tornar redundante.
- **Impacto da Refatoração:** A principal vantagem é a eliminação de código duplicado, o que torna o código resultante menos volumoso, mais fácil de ler, entender e manter. Reduz a confusão para os desenvolvedores, que não precisam mais adivinhar por que existem múltiplas classes fazendo a mesma coisa de formas ligeiramente diferentes.