Categorias de *Code Smells*

1. Bloaters (Exageros / Inchaços)

Os *Bloaters* referem-se a partes do código – sejam métodos, classes ou estruturas de dados – que cresceram desproporcionalmente, tornando-se tão grandes e complexas que são difíceis de gerenciar, entender e modificar. Este tipo de *smell* geralmente não aparece de forma abrupta; em vez disso, acumula-se gradualmente à medida que o software evolui, especialmente quando não há um esforço consciente para controlá-los. O principal impacto dos *Bloaters* é tornar o código maior do que o estritamente necessário, o que, por sua vez, dificulta a manutenção e aumenta a probabilidade de introdução de erros.

1.1 Long Method (Método Longo)

- **Definição:** Um método é considerado longo quando contém um número excessivo de linhas de código. Embora não haja um número mágico, uma heurística comum sugere que métodos com mais de 10 a 15 linhas já deveriam ser examinados com mais atenção. A principal consequência é que, quanto mais longo um método, mais difícil se torna compreendê-lo e mantê-lo.
- **Sintomas e Causas:** A causa mais frequente para a existência de métodos longos é a tendência de adicionar novas funcionalidades a métodos já existentes, simplesmente porque é percebido como mentalmente menos custoso do que criar um novo método para a nova lógica. Outro sintoma claro é a necessidade de inserir comentários dentro do corpo do método para explicar diferentes seções; isso geralmente indica que essas seções poderiam ser extraídas para métodos menores e com nomes autoexplicativos. Um método longo frequentemente viola o Princípio da Responsabilidade Única (SRP) em nível de método, ou seja, ele tenta fazer mais de uma coisa. Embora a contagem de linhas seja um indicador inicial, a coesão funcional é o critério mais importante.
  Métodos longos também podem ser um terreno fértil para outros *code smells*. Por exemplo, a complexidade e o tamanho podem facilmente esconder trechos de código duplicado ou levar à introdução de variáveis locais com nomes pouco claros à medida que o contexto original se perde.
- **Exemplos de Código:**
  Java:
  Um exemplo mostra um método `printOrderDetails` que imprime cabeçalho, itens, endereço de cobrança, endereço de entrega e total, tudo em sequência. Este método é refatorado extraindo cada uma dessas etapas para métodos privados menores como `printHeader`, `printOrderItems`, etc.

*Antes da Refatoração (Exemplo conceitual de `printOrderDetails`):

```
import java.util.Date;
import java.util.List;

class Order { // Definição simplificada
    public String id;
    public Date orderDate;
    public List<String> items;
    public String billingAddress;
    public String shippingAddress;
    public double total;
    // Construtor e outros métodos omitidos
}

public class OrderPrinter {
    public void printOrderDetails(Order order) {
        // Imprimir cabeçalho
        System.out.println("OrderId: " + order.id);
        System.out.println("Order Date: " + order.orderDate);

        // Imprimir itens do pedido
        System.out.println("Items:");
        for (String item : order.items) {
            System.out.println("- " + item);
        }

        // Imprimir endereço de cobrança
        System.out.println("Billing Address: " + order.billingAddress);

        // Imprimir endereço de entrega
        System.out.println("Shipping Address: " + order.shippingAddress);

        // Imprimir total
        System.out.println("Total: $" + order.total);
    }
}
```

Após a Refatoração (Extraindo Métodos):

```java
// Exemplo conceitual baseado em S16
public class OrderPrinterRefactored {
    public void printOrderDetails(Order order) {
        printHeader(order);
        printOrderItems(order);
        printBillingAddress(order);
        printShippingAddress(order);
        printTotal(order);
    }

    private void printHeader(Order order) {
        System.out.println("OrderId: " + order.id);
        System.out.println("Order Date: " + order.orderDate);
    }

    private void printOrderItems(Order order) {
        System.out.println("Items:");
        for (String item : order.items) {
            System.out.println("- " + item);
        }
    }

    private void printBillingAddress(Order order) {
        System.out.println("Billing Address: " + order.billingAddress);
    }

    private void printShippingAddress(Order order) {
        System.out.println("Shipping Address: " + order.shippingAddress);
    }

    private void printTotal(Order order) {
        System.out.println("Total: $" + order.total);
    }
}
```

- **Técnicas de Refatoração Sugeridas:** A principal técnica é **Extrair Método**

(*Extract Method*), que consiste em pegar um fragmento de código dentro do método longo e movê-lo para um novo método (privado ou público, conforme apropriado) e substituir o fragmento original por uma chamada ao novo método. Se o método longo contém condicionais complexas, **Decompor Condicional** (*Decompose Conditional*) pode ser aplicado para extrair as condições e seus blocos para métodos separados. Se loops complexos estão presentes, partes do loop ou o loop inteiro podem ser extraídos. Em casos mais extremos, onde o método é tão complexo que representa uma funcionalidade coesa por si só, pode-se usar **Substituir Método por Objeto de Método** (*Replace Method with Method Object*), transformando o método em sua própria classe onde os parâmetros locais se tornam campos.

- **Impacto da Refatoração:** O código resultante é geralmente mais legível e menor em termos de complexidade por unidade. A refatoração também ajuda a reduzir a duplicação de código, pois métodos menores e mais focados são mais fáceis de reutilizar. Classes com métodos curtos tendem a ser mais robustas e ter uma "vida útil" mais longa no sistema.

1.2. Large Class (Classe Grande / God Class)

- **Definição:** Uma classe é considerada grande, ou uma *God Class*, quando acumula um número excessivo de campos, métodos ou linhas de código.[3] Mais importante do que o tamanho absoluto é a violação do Princípio da Responsabilidade Única (SRP): uma classe grande frequentemente assume muitas responsabilidades que poderiam (e deveriam) ser distribuídas entre classes menores e mais coesas.
- **Sintomas e Causas:** Assim como os métodos longos, as classes grandes geralmente começam pequenas e "incham" com o tempo. Os desenvolvedores tendem a adicionar novas funcionalidades a classes existentes em vez de criar novas classes, pois isso pode parecer mais rápido ou fácil no curto prazo. Uma *God Class* é uma classe que centraliza uma quantidade desproporcional de lógica de negócios ou coordenação de dados, tornando-se um ponto central de dependência e complexidade no sistema.
  A existência de uma *Large Class* frequentemente anda de mãos dadas com *Long Methods*. Se uma classe tem múltiplas e complexas responsabilidades, é natural que seus métodos também se tornem longos para implementar essas responsabilidades. Para um LLM, identificar uma *God Class* é um desafio mais sofisticado do que simplesmente contar linhas de código ou número de métodos. Requer uma análise da coesão dos membros da classe (quão bem seus campos e métodos se agrupam em torno de responsabilidades distintas) e de suas dependências externas. O LLM precisaria aprender a reconhecer esses

agrupamentos lógicos de responsabilidades que poderiam ser candidatos à extração para novas classes.

- **Exemplos de Código:**
  - Java:
    O exemplo descreve uma classe User que armazena não apenas dados básicos do usuário (email, firstName), mas também informações de endereço (address, city, state, zipCode) e perfil (bio), além de métodos para manipular tanto o endereço completo quanto o perfil. Isso sugere pelo menos duas responsabilidades distintas (dados pessoais e dados de endereço/perfil) que poderiam ser separadas.
    Antes da Refatoração (Classe User com múltiplas responsabilidades):

```
public class User {
    private String email;
    private String firstName;
    private String lastName;
    // Campos de endereço
    private String address;
    private String city;
    private String state;
    private int zipCode;
    // Campos de perfil
    private String bio;
    private String profilePictureUrl;

    public User(String email, String firstName, String lastName, String address, String city,
String state, int zipCode, String bio, String profilePictureUrl) {
        this.email = email;
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.bio = bio;
        this.profilePictureUrl = profilePictureUrl;
    }

    // Métodos relacionados ao endereço
    public String getFullAddress() {
```

```
        return address + ", " + city + ", " + state + " " + zipCode;
    }
    public void updateAddress(String address, String city, String state, int zipCode) {
        this.address = address;
        this.city = city;
        //...
    }

    // Métodos relacionados ao perfil
    public String getProfileSummary() {
        return "Name: " + firstName + " " + lastName + "\nBio: " + bio;
    }
    public void updateBio(String bio) {
        this.bio = bio;
    }
    // Getters e Setters omitidos para brevidade
}
```

Após a Refatoração (Extraindo classes Address e Profile):

```
class Address {
    private String street;
    private String city;
    private String state;
    private int zipCode;

    public Address(String street, String city, String state, int zipCode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
    }

    public String getFullAddress() {
        return street + ", " + city + ", " + state + " " + zipCode;
    }
    // Getters e Setters para Address
}

class Profile {
```

```
    private String bio;
    private String profilePictureUrl;

    public Profile(String bio, String profilePictureUrl) {
        this.bio = bio;
        this.profilePictureUrl = profilePictureUrl;
    }

    public String getBio() { return bio; }
    public void setBio(String bio) { this.bio = bio; }
    // Outros getters e setters para Profile
}

public class UserRefactored {
    private String email;
    private String firstName;
    private String lastName;
    private Address address; // Composição
    private Profile profile; // Composição

    public UserRefactored(String email, String firstName, String lastName, Address address,
Profile profile) {
        this.email = email;
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
        this.profile = profile;
    }

    public Address getAddress() { return address; }
    public void setAddress(Address address) { this.address = address; }
    public Profile getProfile() { return profile; }
    public void setProfile(Profile profile) { this.profile = profile; }
    // Getters e Setters para email, firstName, lastName
}
```

- **Técnicas de Refatoração Sugeridas:** A principal técnica é **Extrair Classe** (*Extract Class*), que é usada quando um subconjunto de responsabilidades (dados e métodos) da classe grande pode ser agrupado em um novo

componente coeso e separado. Se parte do comportamento da classe grande pode ser implementada de maneiras diferentes ou é usada apenas em casos raros, **Extrair Subclasse** (*Extract Subclass*) pode ser apropriado, criando uma hierarquia onde a classe original se torna a superclasse. Se o objetivo é definir um contrato claro para os clientes da classe, **Extrair Interface** (*Extract Interface*) pode ser útil, permitindo que diferentes implementações (incluindo a classe grande refatorada ou suas partes extraídas) adiram a essa interface.

- **Impacto da Refatoração:** A refatoração de classes grandes alivia os desenvolvedores da necessidade de lembrar um grande número de atributos e métodos para uma única classe. Frequentemente, a divisão de classes grandes ajuda a evitar a duplicação de código e funcionalidade, pois responsabilidades mais claras podem levar à identificação de componentes reutilizáveis. O resultado é um sistema com maior coesão dentro das classes e menor acoplamento entre elas.

1.3. Primitive Obsession (Obsessão por Tipos Primitivos)

- **Definição:** Este *smell* ocorre quando tipos de dados primitivos (como inteiros, strings, booleanos, arrays simples) são usados excessivamente para representar conceitos de domínio que seriam mais bem encapsulados por pequenos objetos ou classes dedicadas.
- **Sintomas e Causas:** A principal causa é a aparente simplicidade de adicionar um novo campo primitivo em vez de projetar e criar uma nova classe, especialmente sob pressão de prazos. Sintomas incluem o uso de múltiplas variáveis primitivas que, juntas, representam um único conceito (e.g., day, month, year para uma data), o uso de constantes para codificar informações de tipo ou estado (e.g., ORDER_STATUS_PENDING = 1), ou o uso de strings como nomes de campos em arrays de dados. Uma consequência direta é a falta de encapsulamento e a dispersão da lógica de validação e manipulação desses "tipos simulados" por todo o código.
  A obsessão por primitivos frequentemente leva a que a lógica de validação e as regras de negócio associadas a esses dados fiquem espalhadas, violando o princípio DRY (Don't Repeat Yourself) e tornando essas regras implícitas e difíceis de gerenciar e modificar consistentemente. Para um LLM, o desafio é reconhecer não apenas o uso de um primitivo, mas identificar grupos de primitivos que conceitualmente formam uma unidade e que possuem comportamento ou regras de validação associados que poderiam ser encapsulados.
- **Exemplos de Código:**
  Java:
  Exemplos conceituais incluem o uso de `String` para CEP (`ZipCode`), `String`

para `UserName` ou `EmailAddress`, ou `int`/`double` para representar dinheiro (`Money`), todos os quais se beneficiariam de classes dedicadas que encapsulam tanto o valor quanto as operações e validações associadas.

- **Técnicas de Refatoração Sugeridas:** A principal técnica é **Substituir Valor de Dado por Objeto** (*Replace Data Value with Object*), que envolve criar uma nova classe para encapsular o valor primitivo e seu comportamento associado.[7] Se os primitivos são frequentemente passados juntos como parâmetros de método, **Introduzir Objeto de Parâmetro** (*Introduce Parameter Object*) ou **Preservar Objeto Inteiro** (*Preserve Whole Object*) podem ser usados para agrupá-los.[7] Quando primitivos são usados para simular tipos (e.g., códigos de status), técnicas como **Substituir Código de Tipo por Classe** (*Replace Type Code with Class*), **Subclasses** (*Replace Type Code with Subclasses*) ou **Estado/Estratégia** (*Replace Type Code with State/Strategy*) são aplicáveis.[7] Se um array está sendo usado para armazenar uma coleção heterogênea de dados primitivos, **Substituir Array por Objeto** (*Replace Array with Object*) é a refatoração indicada.[7]
- **Impacto da Refatoração:** O código torna-se mais flexível devido ao uso de objetos em vez de primitivos. Há uma melhoria na organização e na compreensibilidade, pois as operações e validações relacionadas a um dado específico ficam centralizadas em sua própria classe, em vez de espalhadas pelo sistema.[7] Isso também facilita a identificação de código duplicado. O modelo de domínio do software torna-se mais explícito e robusto.

1.4. Long Parameter List (Lista Longa de Parâmetros)

- **Definição:** Um método ou construtor apresenta uma lista longa de parâmetros quando recebe um número excessivo de argumentos. Embora não exista um limite rígido, geralmente considera-se que mais de três ou quatro parâmetros já indicam um possível *smell*.
- **Sintomas e Causas:** Listas longas de parâmetros podem surgir da fusão de múltiplos algoritmos em um único método, onde os parâmetros controlam qual variação do algoritmo é executada. Outra causa comum é a tentativa de reduzir o acoplamento entre classes, passando todas as dependências necessárias como parâmetros, em vez de a classe obter essas dependências internamente. Com o tempo, à medida que mais informações são necessárias, mais parâmetros são adicionados, tornando a lista difícil de entender, usar e manter, e propensa a inconsistências.
  Uma lista longa de parâmetros é frequentemente um sintoma de *Data Clumps* não identificados que estão sendo passados como argumentos individuais. Se um conjunto de parâmetros sempre aparece junto em diferentes chamadas de

método, eles provavelmente formam um conceito coeso que poderia ser encapsulado em um objeto. Além disso, testar métodos com listas extensas de parâmetros é mais complexo, exigindo a configuração de muitos valores de entrada e tornando os testes verbosos e frágeis.

- **Exemplos de Código:**
  - Java:
    O exemplo em 10 ilustra um construtor da classe Order que recebe múltiplos detalhes do cliente e do pedido como parâmetros individuais.
    Antes da Refatoração:

```
Java
// Exemplo conceitual baseado em [10]
public class Order {
    public Order(String customerName, String customerAddress, String customerCity, String customerState, String customerZip,
                 int orderNumber, String orderType, String orderDate, String deliveryDate) {
        //... lógica do construtor...
    }
}
```

    Após a Refatoração (Usando Objeto de Parâmetro Customer e, potencialmente, OrderDetails):

```
Java
// Exemplo conceitual baseado em [10]
public class Customer {
    //... campos e construtor para detalhes do cliente...
    public Customer(String name, String address, String city, String state, String zip) { /*... */ }
}

public class OrderRefactored {
    public OrderRefactored(Customer customer, int orderNumber, String orderType, String orderDate, String deliveryDate) {
        //... lógica do construtor...
    }
}
// Uso:
// Customer customer = new Customer("John Doe", "123 Main St", "Anytown", "CA", "90210");
// Order order = new OrderRefactored(customer, 1001, "Online", "2024-01-01", "2024-01-05");
```

    O exemplo em mostra um método foo(author, commit_id, files, sha_id, time) que é refatorado para foo(Commit commit) usando uma dataclass Commit

(conceito aplicável a Java com uma classe Commit).

Técnicas de Refatoração Sugeridas:

- **Substituir Parâmetro por Chamada de Método** (*Replace Parameter with Method Call*): Se um argumento é o resultado de uma chamada de método em outro objeto, e esse objeto está acessível, pode-se chamar o método diretamente dentro do corpo do método que está sendo refatorado, em vez de passar o valor como parâmetro.
- **Preservar Objeto Inteiro** (*Preserve Whole Object*): Se vários parâmetros são obtidos de um mesmo objeto, em vez de extrair esses valores e passá-los individualmente, passe o objeto inteiro como parâmetro.
- **Introduzir Objeto de Parâmetro** (*Introduce Parameter Object*): Esta é uma das técnicas mais comuns. Agrupe os parâmetros que logicamente pertencem juntos em uma nova classe (ou struct/dataclass) e passe uma instância dessa classe como um único parâmetro.[10]
- **Remover Argumento de Flag** (*Remove Flag Argument*): Se um parâmetro booleano é usado para alterar o comportamento do método, considere dividir o método em dois métodos separados, cada um representando um dos comportamentos.

- **Impacto da Refatoração:** O código torna-se mais legível e as assinaturas dos métodos ficam mais curtas e fáceis de entender. Esta refatoração também pode ajudar a revelar código duplicado que não era óbvio anteriormente, pois os novos objetos de parâmetro podem ser reutilizados.

1.5. Data Clumps (Aglomerados de Dados)

- **Definição:** *Data Clumps* são grupos de variáveis (campos, parâmetros de método) que frequentemente aparecem juntas em diferentes partes do código. Exemplos comuns incluem coordenadas (x, y, z), datas de início e fim, ou um conjunto de parâmetros de conexão a um banco de dados. A presença desses "aglomerados" sugere que eles deveriam ser encapsulados em sua própria classe ou estrutura.
- **Sintomas e Causas:** A principal causa de *Data Clumps* é frequentemente uma estrutura de programa deficiente ou o resultado de "programação copy-paste", onde conjuntos de variáveis relacionadas são repetidos em vez de abstraídos. Um bom teste para identificar um *Data Clump* é verificar se, ao remover um dos valores do grupo, os outros perdem parte de seu significado contextual. Se isso acontecer, é um forte indicativo de que o grupo de variáveis forma uma unidade conceitual coesa.

É importante notar que a refatoração de *Data Clumps* para uma nova classe corre o risco de criar um *Data Class* (outro *code smell*, discutido na seção 2.4.3) se o comportamento associado aos dados não for movido junto com eles para a nova classe. A criação da classe de dados é apenas o primeiro passo; a funcionalidade que opera nesses dados deve, idealmente, residir nessa nova classe para aumentar a coesão.

- **Exemplos de Código:**
  - Java:
    O exemplo em e mostra um método welcomeNew(firstName, lastName, age, gender, occupation, city). Todos esses parâmetros formam um Data Clump representando uma pessoa.
    Antes da Refatoração:

```
public class OldSystem {
    public static void welcomeNew(String firstName, String lastName, int age, String gender,
String occupation, String city) {
        System.out.printf("Welcome %s %s, a %d-year-old %s from %s who works as a
%s%n",
                          firstName, lastName, age, gender, city, occupation);
    }

    public static void main(String args) {
        String fName = "John";
        String lName = "Doe";
        int userAge = 30;
        String userGender = "Male";
        String userOccupation = "Engineer";
        String userCity = "Anytown";
        welcomeNew(fName, lName, userAge, userGender, userOccupation,
userCity);
        // Outros métodos poderiam usar o mesmo clump:
        // registerUser(fName, lName, userAge, userGender, userOccupation, userCity);
        // updateUserProfile(fName, lName, userAge, userGender, userOccupation, userCity);
    }
}
```

    Após a Refatoração (Introduzindo a classe Person):

```
class Person {
    String firstName;
```

```
    String lastName;
    int age;
    String gender;
    String occupation;
    String city;

    public Person(String firstName, String lastName, int age, String gender, String occupation,
String city) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.gender = gender;
        this.occupation = occupation;
        this.city = city;
    }

    public void welcomeNew() { // Comportamento movido para a classe Person
        System.out.printf("Welcome %s %s, a %d-year-old %s from %s who works as a
%s%n",
                          firstName, lastName, age, gender, city, occupation);
    }
    // Outros métodos relevantes para Person poderiam ser adicionados aqui
}

public class NewSystem {
    public static void main(String args) {
        Person joe = new Person("John", "Doe", 30, "Male", "Engineer", "Anytown");
        joe.welcomeNew();
        // Outros métodos agora usariam o objeto Person:
        // registerUser(joe);
        // updateUserProfile(joe);
    }
}
```

- **Técnicas de Refatoração Sugeridas:**
  - **Extrair Classe** (*Extract Class*): Se os dados que se repetem são campos de uma ou mais classes, eles podem ser movidos para sua própria classe dedicada.
  - **Introduzir Objeto de Parâmetro** (*Introduce Parameter Object*): Se os

mesmos grupos de dados são consistentemente passados como parâmetros para métodos, esses parâmetros podem ser agrupados em uma nova classe, e uma instância dessa classe é passada como um único argumento.
    - **Preservar Objeto Inteiro** (*Preserve Whole Object*): Se você está extraindo alguns valores de um objeto para passá-los como parâmetros para um método, e esses valores formam um *Data Clump* (ou mesmo se não formam, mas o método poderia logicamente operar no objeto inteiro), considere passar o objeto original inteiro em vez dos campos individuais.
    - É fundamental também **mover o comportamento** que opera nesses dados para a nova classe de dados, para evitar a criação de uma *Data Class*.
- **Impacto da Refatoração:** A refatoração de *Data Clumps* melhora significativamente a compreensão e a organização do código, pois as operações sobre dados específicos ficam centralizadas em um único local. Isso também tende a reduzir o tamanho geral do código, especialmente diminuindo a verbosidade das listas de parâmetros.

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.

3. Change Preventers (Prevenidores de Mudanças)

Os *Change Preventers* são *code smells* que tornam a modificação e evolução do software particularmente difíceis e arriscadas. Eles se manifestam quando uma alteração aparentemente simples em um local do código exige uma cascata de modificações em muitos outros lugares, muitas vezes não relacionados diretamente. Isso torna o processo de desenvolvimento mais lento, caro e propenso a erros, pois é fácil esquecer alguma das alterações necessárias ou introduzir efeitos colaterais indesejados. Esses *smells* geralmente indicam um alto acoplamento entre diferentes partes do sistema e uma falta de separação de responsabilidades.

3.1. Divergent Change (Mudança Divergente)

- **Definição:** Ocorre quando uma única classe é frequentemente alterada de maneiras diferentes e por razões diferentes e não relacionadas.[19] É o oposto do *smell Shotgun Surgery*, onde uma única mudança conceitual afeta múltiplas classes.
- **Sintomas e Causas:** O principal sintoma é a necessidade de modificar múltiplos métodos não relacionados dentro de uma mesma classe sempre que um novo requisito surge ou uma alteração externa ocorre.[19] Por exemplo, ao adicionar um novo tipo de produto a um sistema de e-commerce, pode ser necessário alterar métodos na mesma classe Product para lidar com cálculo de preço, exibição, validação de estoque e formatação de relatório para esse novo tipo. Isso é uma clara violação do Princípio da Responsabilidade Única (SRP), pois a classe está acumulando múltiplas responsabilidades que mudam por diferentes eixos. Identificar *Divergent Change* pode, por vezes, ser auxiliado pela análise do histórico de mudanças do sistema de controle de versão. Se uma classe específica é consistentemente alterada em *commits* que descrevem modificações em funcionalidades conceitualmente distintas, isso é um forte indicador do *smell*.
- **Exemplos de Código:**
  - Java:
    O exemplo em e descreve uma classe Patient que, inicialmente, contém tanto os dados do paciente quanto a lógica para exportar o histórico de tratamento para CSV. Se for necessário adicionar um novo formato de exportação (e.g., JSON) ou se a forma de armazenar os dados do paciente mudar (e.g., adicionar número de seguro), a classe Patient precisará ser alterada por duas razões distintas.
    Antes da Refatoração (Classe Patient com múltiplas responsabilidades):
    ```
    using System.IO;
    using System.Text;
    ```

```
using System.Collections.Generic;

public class Patient
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public List<string> TreatmentHistory { get; private set; }

    public Patient(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
        TreatmentHistory = new List<string>();
    }

    public void AddTreatment(string treatment)
    {
        TreatmentHistory.Add(treatment);
    }

    // Responsabilidade 1: Gerenciar dados do paciente (acima)
    // Responsabilidade 2: Exportar histórico de tratamento (abaixo)
    public string ExportTreatmentHistoryToCsv()
    {
        var csvBuilder = new StringBuilder();
        csvBuilder.AppendLine("TreatmentDate,Description,Doctor"); // Exemplo de
cabeçalho
        foreach (var treatment in TreatmentHistory)
        {
            // Supondo que 'treatment' é uma string formatada "data,desc,medico"
            csvBuilder.AppendLine(treatment);
        }
        return csvBuilder.ToString();
    }
    // Se precisarmos exportar para JSON, adicionamos outro método aqui,
    // modificando a classe Patient por uma nova razão (formato de exportação).
    // Se precisarmos adicionar InsuranceNumber, modificamos a classe Patient
    // por outra razão (dados do paciente).
}
```

Após a Refatoração (Extraindo a responsabilidade de exportação):

C#

```
// Exemplo baseado em S216, S256
public class PatientRefactored
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public string InsuranceNumber { get; private set; } // Nova informação do paciente
    public List<string> TreatmentHistory { get; private set; }

    public PatientRefactored(string firstName, string lastName, string insuranceNumber)
    {
        FirstName = firstName;
        LastName = lastName;
        InsuranceNumber = insuranceNumber;
        TreatmentHistory = new List<string>();
    }

    public void AddTreatment(string treatment)
    {
        TreatmentHistory.Add(treatment);
    }
}

public interface ITreatmentHistoryExporter
{
    string Export(List<string> treatments);
}

public class CsvTreatmentHistoryExporter : ITreatmentHistoryExporter
{
    public string Export(List<string> treatments)
    {
        var csvBuilder = new StringBuilder();
        csvBuilder.AppendLine("TreatmentDate,Description,Doctor");
        foreach (var treatment in treatments)
        {
            csvBuilder.AppendLine(treatment);
```

```
        }
        return csvBuilder.ToString();
    }
}

public class JsonTreatmentHistoryExporter : ITreatmentHistoryExporter
{
    public string Export(List<string> treatments)
    {
        // Lógica para converter 'treatments' para formato JSON
        Console.WriteLine("Exporting treatments to JSON format.");
        return "{\"treatments\":}"; // Exemplo simplificado
    }
}
```

Agora, a classe PatientRefactored muda apenas por razões relacionadas aos dados do paciente. A lógica de exportação está em classes separadas (CsvTreatmentHistoryExporter, JsonTreatmentHistoryExporter), que mudam apenas se o formato de exportação mudar.

- **Técnicas de Refatoração Sugeridas:** A principal abordagem é **Extrair Classe** (*Extract Class*), dividindo a classe original em duas ou mais classes, cada uma com um conjunto coeso de responsabilidades. Se o problema envolve diferentes tipos de objetos sendo tratados de forma similar, mas com variações que causam mudanças divergentes na classe principal, técnicas de herança como **Extrair Superclasse** (*Extract Superclass*) ou **Extrair Subclasse** (*Extract Subclass*) podem ser usadas para melhor organizar o comportamento comum e variado.
- **Impacto da Refatoração:** Melhora a organização do código ao garantir que cada classe tenha uma única razão para mudar, alinhando-se com o SRP. Isso reduz a duplicação de lógica de decisão e simplifica o suporte e a evolução do sistema, pois as mudanças se tornam mais localizadas.

3.2. Shotgun Surgery (Cirurgia de Espingarda)

- **Definição:** Oposto ao *Divergent Change*, *Shotgun Surgery* ocorre quando uma única mudança conceitual em uma funcionalidade requer a aplicação de muitas pequenas alterações em múltiplas classes diferentes simultaneamente.
- **Sintomas e Causas:** A causa raiz é frequentemente a falta de abstração adequada e a duplicação de lógica comum, que acaba espalhada por várias

classes (código copiado e colado). Também pode ser resultado de uma violação excessiva do Princípio da Responsabilidade Única, onde uma única responsabilidade foi fragmentada desnecessariamente em muitas classes pequenas, ou de um acoplamento excessivo entre classes.
Este *smell* é frequentemente um resultado direto da violação do princípio DRY (Don't Repeat Yourself). A lógica que deveria ser centralizada e reutilizada acaba sendo espalhada, e qualquer modificação nessa lógica exige encontrar e alterar todas as suas cópias. Além disso, *Message Chains* longas também podem levar a *Shotgun Surgery*, pois uma mudança em qualquer elo da cadeia pode propagar a necessidade de alterações no código cliente e em outros elos da cadeia.

- **Exemplos de Código:**
  - Java:
    O exemplo em 20 e descreve uma classe SavingsAccount onde a lógica de validação do saldo mínimo (e.g., balance < 1000) está duplicada nos métodos withdraw e transfer. Se a regra do saldo mínimo mudar, ambas as classes (e potencialmente outras que usem a mesma lógica) precisarão ser alteradas.
    Antes da Refatoração (Lógica de validação duplicada):

```
Java
// Exemplo baseado em [20], S126
public class SavingsAccount {
    private double balance;
    private static final double MINIMUM_BALANCE_THRESHOLD = 1000;

    public SavingsAccount(double initialBalance) { this.balance = initialBalance; }

    public void withdraw(double amount) {
        if (this.balance - amount < MINIMUM_BALANCE_THRESHOLD) { // Lógica duplicada
            throw new IllegalArgumentException("Withdrawal would bring balance below threshold!");
        }
        this.balance -= amount;
        System.out.println("Withdrew " + amount);
    }

    public void transfer(double amount, SavingsAccount otherAccount) {
        if (this.balance - amount < MINIMUM_BALANCE_THRESHOLD) { // Lógica duplicada
            throw new IllegalArgumentException("Transfer would bring balance below threshold!");
```

```
        }
        this.balance -= amount;
        otherAccount.deposit(amount); // Supondo um método deposit
        System.out.println("Transferred " + amount);
    }
    public void deposit(double amount) { this.balance += amount; } // Método auxiliar
}
// Se a regra de MINIMUM_BALANCE_THRESHOLD ou a forma de checá-la mudar,
// ambos os métodos withdraw e transfer precisam ser alterados.
```

Após a Refatoração (Extraindo método validateTransaction):

```
Java
// Exemplo baseado em [20], S126
public class SavingsAccountRefactored {
    private double balance;
    private static final double MINIMUM_BALANCE_THRESHOLD = 1000;

    public SavingsAccountRefactored(double initialBalance) { this.balance =
initialBalance; }

    private void validateTransaction(double amountToDebit) { // Método extraído
        if (this.balance - amountToDebit < MINIMUM_BALANCE_THRESHOLD) {
            throw new IllegalArgumentException("Transaction would bring balance below
threshold!");
        }
    }

    public void withdraw(double amount) {
        validateTransaction(amount); // Chamada ao método centralizado
        this.balance -= amount;
        System.out.println("Withdrew " + amount);
    }

    public void transfer(double amount, SavingsAccountRefactored otherAccount) {
        validateTransaction(amount); // Chamada ao método centralizado
        this.balance -= amount;
        otherAccount.deposit(amount);
        System.out.println("Transferred " + amount);
    }
    public void deposit(double amount) { this.balance += amount; }
```

```
}
```

Agora, se a lógica de validação do saldo mínimo precisar mudar, apenas o método validateTransaction precisará ser alterado.

- Geral (Logging):
  Um exemplo canônico é a adição de funcionalidade de logging. Se for necessário logar a entrada e saída de múltiplas funções em diferentes classes, e isso for feito adicionando manualmente chamadas de log.debug("Entering X") e log.debug("Exiting X") em cada uma, qualquer mudança no formato do log ou no nível de log exigirá a edição de todas essas funções.

- **Técnicas de Refatoração Sugeridas:** A principal estratégia é consolidar a funcionalidade espalhada. Isso pode ser feito usando **Mover Método** (*Move Method*) e **Mover Campo** (*Move Field*) para agrupar partes relacionadas da lógica em uma única classe ou em um número menor de classes. Se a *Shotgun Surgery* é causada por duplicação de código, **Extrair Método** (*Extract Method*) ou **Extrair Classe** (*Extract Class*) pode ser usado para criar um componente compartilhado que encapsula a lógica duplicada. Em alguns casos, se uma classe intermediária se tornou muito pequena e apenas delega chamadas, **Juntar Classe em Linha** (*Inline Class*) pode ser considerado para remover a indireção.
- **Impacto da Refatoração:** Reduz drasticamente o número de classes que precisam ser tocadas para uma única mudança conceitual. Melhora a modularidade, coesão e reduz o acoplamento, tornando o código mais fácil de entender e manter.

3.3. Parallel Inheritance Hierarchies (Hierarquias de Herança Paralelas)

- **Definição:** Este *smell* ocorre quando, para cada subclasse criada em uma hierarquia de herança, é necessário criar uma subclasse correspondente em outra hierarquia de herança. As duas (ou mais) hierarquias evoluem em paralelo, espelhando uma à outra.
- **Sintomas e Causas:** Um sintoma comum é a presença de um prefixo ou sufixo comum nos nomes das classes em ambas as hierarquias (e.g., Vehicle e VehicleView, com subclasses Car/CarView, Truck/TruckView). O problema fundamental é que essa estrutura duplicada exige que qualquer adição ou modificação em uma hierarquia seja refletida na outra, aumentando o esforço de manutenção e o risco de inconsistências. Isso indica um forte acoplamento estrutural e uma forma de duplicação conceitual, onde a lógica de variação está replicada em múltiplas árvores de herança. A necessidade de manter essas hierarquias sincronizadas manualmente é propensa a erros.

- Exemplos de Código:
  O material de pesquisa é primariamente conceitual para este smell, sem fornecer exemplos de código "antes e depois" completos e refatorados.
  Conceitual:
  Imagine uma hierarquia de formas geométricas e uma hierarquia paralela para seus respectivos renderizadores:

```
Java
// Hierarquia 1: Formas
abstract class Shape { abstract void draw(); }
class Circle extends Shape { @Override void draw() { /* desenha círculo */ } }
class Square extends Shape { @Override void draw() { /* desenha quadrado */ } }

// Hierarquia 2: Renderizadores (paralela e acoplada)
abstract class ShapeRenderer { abstract void render(Shape s); } // Problema: Shape s
class CircleRenderer extends ShapeRenderer { @Override void render(Shape s) { if (s instanceof Circle) { /* renderiza círculo */ } } }
class SquareRenderer extends ShapeRenderer { @Override void render(Shape s) { if (s instanceof Square) { /* renderiza quadrado */ } } }
```

  Se uma nova forma Triangle é adicionada, um TriangleRenderer também precisa ser criado. A refatoração buscaria quebrar essa dependência paralela. Uma abordagem seria usar o padrão Visitor, ou fazer com que Shape tivesse um método getRenderer() que retornasse uma instância de um IRenderer apropriado, ou que o próprio Shape soubesse como se renderizar (se a responsabilidade de renderização pertencer à forma).
  Refatoração Conceitual (Simplificada - movendo responsabilidade):

```
Java
// Hierarquia única com responsabilidade de renderização
abstract class ShapeRefactored {
    abstract void renderSelf(); // Cada forma sabe como se renderizar
}
class CircleRefactored extends ShapeRefactored {
    @Override void renderSelf() { System.out.println("Rendering Circle"); }
}
class SquareRefactored extends ShapeRefactored {
    @Override void renderSelf() { System.out.println("Rendering Square"); }
}
// Não há mais hierarquia de ShapeRenderer.
// O cliente simplesmente chama shape.renderSelf();
```

- **Técnicas de Refatoração Sugeridas:** A estratégia geral é quebrar a dependência entre as hierarquias paralelas. Frequentemente, isso envolve fazer com que as instâncias de uma hierarquia referenciem instâncias da outra, em vez de espelhar a estrutura de herança. **Mover Método** (*Move Method*) e **Mover Campo** (*Move Field*) podem ser usados para consolidar responsabilidades e reduzir a necessidade de classes paralelas. O padrão de projeto **Bridge** é especificamente projetado para desacoplar uma abstração de sua implementação, permitindo que ambas evoluam independentemente, o que pode ser uma solução robusta para este *smell*. Em outros casos, pode-se fundir as hierarquias se uma delas for apenas um conjunto de dados ou comportamentos que podem ser incorporados na outra.
- **Impacto da Refatoração:** Reduz a duplicação estrutural e a necessidade de modificações em cascata ao estender uma das hierarquias. Simplifica a adição de novas variantes, pois a mudança fica mais localizada. Melhora a manutenibilidade geral do sistema.

4. Dispensables (Dispensáveis)

Os *Dispensables* são elementos do código que são desnecessários, redundantes ou que fornecem pouco valor real, e cuja remoção tornaria o código mais limpo, mais eficiente e mais fácil de entender e manter.[21] Eles representam um "entulho" que pode obscurecer a lógica importante, aumentar a carga cognitiva dos desenvolvedores e, em alguns casos, até mesmo levar a erros se o código dispensável estiver desatualizado ou for enganoso.

4.1. Comments (Comentários Excessivos ou Obsoletos)

- **Definição:** Este *smell* ocorre quando um método, classe ou bloco de código está repleto de comentários explicativos que são, na verdade, redundantes, obsoletos, enganosos, ou que tentam justificar um código confuso que deveria ser refatorado para ser autoexplicativo.[21]
- **Sintomas e Causas:** Comentários são frequentemente criados com as melhores intenções, geralmente quando o autor do código percebe que sua lógica não é imediatamente óbvia ou intuitiva. No entanto, nesses casos, os comentários podem funcionar como um "desodorante" que mascara o "mau cheiro" de um código que precisa de melhorias estruturais. O ideal é que o próprio código seja claro o suficiente, e o melhor comentário é, muitas vezes, um nome bem escolhido para um método, variável ou classe. Outros sintomas incluem blocos de código comentado (que deveriam estar em um sistema de controle de versão) e comentários que meramente parafraseiam o que uma linha de código já diz de forma óbvia.

Comentários frequentemente mascaram um problema subjacente no código. Em vez de comentar código complexo, o foco deveria ser em simplificar o próprio código. Além disso, código comentado deixado para trás representa uma forma de débito técnico; ele polui a base de código, pode rapidamente se tornar desatualizado em relação ao código ativo ao redor e confunde os desenvolvedores que o encontram posteriormente. Sistemas de controle de versão são a ferramenta apropriada para rastrear e recuperar código antigo, não blocos comentados.

- **Exemplos de Código:**

Java: mostra comentários excessivos em um método `CalculateTriangleArea` que explica a fórmula da área do triângulo, que é autoevidente pelo nome do método e pelos parâmetros. A refatoração envolve remover os comentários internos e, possivelmente, manter um único comentário de documentação no nível do método, se necessário.

- **Técnicas de Refatoração Sugeridas:**
  - Se um comentário está explicando uma expressão complexa, essa expressão deve ser quebrada em subexpressões menores e atribuída a variáveis com nomes autoexplicativos usando **Extrair Variável** (*Extract Variable*).
  - Se um comentário descreve um bloco de código, esse bloco deve ser extraído para um novo método com um nome descritivo (muitas vezes, o próprio texto do comentário pode inspirar o nome do novo método) usando **Extrair Método** (*Extract Method*).
  - Se um método já foi extraído, mas ainda necessita de comentários para explicar o que faz, o método em si deve ser renomeado para ser autoexplicativo usando **Renomear Método** (*Rename Method*).
  - Comentários redundantes, obsoletos ou código comentado devem ser simplesmente removidos. Se houver receio de perder código comentado, deve-se confiar no sistema de controle de versão.
  - Para regras sobre o estado necessário do sistema, em vez de comentários, pode-se usar **Introduzir Asserção** (*Introduce Assertion*).
- **Impacto da Refatoração:** O código torna-se mais limpo, mais conciso e, idealmente, autoexplicativo, reduzindo o ruído visual e a carga cognitiva para os desenvolvedores. A confiança na documentação (que agora é o próprio código) aumenta.

4.2. Duplicate Code (Código Duplicado)

- **Definição:** Este é um dos *code smells* mais comuns e prejudiciais. Ocorre quando

o mesmo fragmento de código, ou fragmentos muito semelhantes, aparecem em mais de um lugar na base de código.[21]

- **Sintomas e Causas:** A duplicação pode surgir quando múltiplos programadores trabalham em partes diferentes do mesmo sistema sem comunicação adequada, resultando na reimplementação de lógica similar. Pressão por prazos pode levar à prática de copiar e colar código existente que é "quase" o que se precisa, com pequenas modificações, em vez de criar uma abstração reutilizável. Em alguns casos, pode ser simplesmente resultado de descuido ou preguiça. A duplicação nem sempre é uma cópia exata; pode ser sutil, com variações na nomeação de variáveis ou pequenas diferenças na lógica, tornando-a mais difícil de detectar. O principal problema com código duplicado é que qualquer bug encontrado na lógica duplicada precisa ser corrigido em todos os locais onde o código foi copiado. Da mesma forma, qualquer alteração ou melhoria nessa lógica precisa ser replicada consistentemente, o que é um processo propenso a erros e omissões, levando a inconsistências e comportamento divergente no sistema.
- **Exemplos de Código:**
  - Java:
    mostra duas linhas de código calculando preços com um desconto fixo: int totalApplesPrice = quantityApples * priceApple - 5; e int totalBananasPrice = quantityBananas * priceBanana - 5;. A parte * price - 5 é duplicada.
    Antes da Refatoração:

```
Java
// Exemplo baseado em S14
// Supondo que quantityApples, priceApple, quantityBananas, priceBanana são definidos
// int totalApplesPrice = quantityApples * priceApple - 5;
// int totalBananasPrice = quantityBananas * priceBanana - 5;
```

    Após a Refatoração (Extraindo método CalculatePrice):

```
Java
// Exemplo baseado em S14
public static class PriceCalculator
{
    private const int Discount = 5;

    public static int CalculatePriceWithDiscount(int quantity, int itemPrice)
    {
        return quantity * itemPrice - Discount;
    }
}
// Uso:
// int totalApplesPrice = PriceCalculator.CalculatePriceWithDiscount(quantityApples,
```

```
priceApple);
// int totalBananasPrice = PriceCalculator.CalculatePriceWithDiscount(quantityBananas,
priceBanana);
```

- **Técnicas de Refatoração Sugeridas:** A abordagem geral é "Don't Repeat Yourself" (DRY).
  - Se o código duplicado está em dois ou mais métodos da mesma classe: use **Extrair Método** (*Extract Method*) e chame o novo método de ambos os locais.
  - Se o código duplicado está em subclasses do mesmo nível hierárquico: use **Extrair Método** em ambas as classes e, em seguida, considere **Puxar Método para Cima** (*Pull Up Method*) para a superclasse. Se o método usa campos das subclasses, pode ser necessário **Puxar Campo para Cima** (*Pull Up Field*) também. Se a duplicação está em construtores, use **Puxar Corpo do Construtor para Cima** (*Pull Up Constructor Body*). Se o código é similar mas não idêntico, **Formar Método Modelo** (*Form Template Method*) pode ser uma solução, onde a estrutura geral fica na superclasse e as variações são implementadas nas subclasses.
  - Se o código duplicado está em classes não relacionadas: Se as classes compartilham uma responsabilidade comum que pode ser abstraída, use **Extrair Superclasse** (*Extract Superclass*). Se criar uma superclasse não é viável ou apropriado, use **Extrair Classe** (*Extract Class*) para mover a funcionalidade duplicada para uma nova classe auxiliar, que pode então ser usada por ambas as classes originais.
  - Se a duplicação ocorre dentro de expressões condicionais: **Consolidar Expressão Condicional** (*Consolidate Conditional Expression*) ou **Consolidar Fragmentos Condicionais Duplicados** (*Consolidate Duplicate Conditional Fragments*) podem ajudar a simplificar e remover a redundância.
- **Impacto da Refatoração:** A fusão de código duplicado simplifica a estrutura do código e o torna mais curto e conciso. Um código mais simples e curto é inerentemente mais fácil de entender e mais barato de manter, pois as modificações na lógica só precisam ser feitas em um único lugar.

4.3. Data Class (Classe de Dados)

- **Definição:** Uma *Data Class* é uma classe que possui principalmente campos (atributos) e métodos triviais para acessá-los (getters e setters), mas carece de comportamento ou funcionalidade significativa própria.[21] Essas classes atuam essencialmente como contêineres passivos de dados, com outras classes assumindo a responsabilidade de manipular esses dados.

- **Sintomas e Causas:** É comum que classes recém-criadas comecem com apenas alguns campos públicos ou propriedades com getters/setters. O *smell* surge quando a classe permanece nesse estado, e a lógica que opera sobre seus dados reside em outras classes. O verdadeiro poder dos objetos reside na combinação de dados e comportamento; uma *Data Class* falha em realizar esse potencial. *Data Classes* frequentemente levam ao *smell Feature Envy* (discutido na seção 2.5.1) em outras classes. Se uma classe Order é puramente dados, uma classe OrderProcessor provavelmente terá métodos que acessam e manipulam muitos campos de Order para realizar seus cálculos, demonstrando "inveja" dos dados de Order. Mover o comportamento relevante para a classe Order pode resolver ambos os *smells*. É importante notar, no entanto, que nem toda classe que primariamente contém dados é um *smell*. Em certos contextos arquiteturais, como Data Transfer Objects (DTOs) usados para transferir dados entre camadas, ou entidades simples em alguns ORMs, classes focadas em dados são intencionais e aceitáveis. O *smell* real ocorre quando há comportamento que *deveria* estar encapsulado junto com os dados na *Data Class*, mas está espalhado por outras partes do sistema.
- Exemplos de Código:
  O material de pesquisa foca mais na definição. Um exemplo concreto seria:
  Conceitual (Java):
  Antes da Refatoração:

```
Java
// Data Class
public class CustomerData {
    public String name;
    public String email;
    public String address;
    // Getters e Setters para todos os campos
}

// Lógica de negócio em outra classe
public class CustomerService {
    public boolean isValidCustomer(CustomerData customer) {
        // Validação do nome, email, endereço
        if (customer.name == null |
```

| customer.name.isEmpty()) return false;
if (customer.email == null ||!customer.email.contains("@")) return false;
//... mais validações

```
return true;
}

    public String getMailingLabel(CustomerData customer) {
        return customer.name + "\n" + customer.address;
    }
}
```

Após a Refatoração (Movendo comportamento para `CustomerData`):
```java
public class Customer { // Renomeada e com comportamento
    private String name;
    private String email;
    private String address;

    public Customer(String name, String email, String address) {
        // Validação pode ocorrer aqui ou em setters
        this.name = name;
        this.email = email;
        this.address = address;
    }

    // Getters e Setters (podem incluir validação)

    public boolean isValid() { // Comportamento agora na classe
        if (name == null |

| name.isEmpty()) return false;
if (email == null ||!email.contains("@")) return false;
//... mais validações
return true;
}

    public String getMailingLabel() { // Comportamento agora na classe
        return name + "\n" + address;
    }
}

// CustomerService agora pode ser mais simples ou até desnecessária para essas
```

```
operações
public class CustomerServiceRefactored {
   public void processCustomer(Customer customer) {
      if (customer.isValid()) {
         //...
         String label = customer.getMailingLabel();
         //...
      }
   }
}
```

- **Técnicas de Refatoração Sugeridas:**
  - **Encapsular Campos** (*Encapsulate Field*): Se a classe possui campos públicos, o primeiro passo é torná-los privados e fornecer getters e setters públicos para controlar o acesso.
  - **Mover Método** (*Move Method*): A principal refatoração é identificar comportamentos no código cliente que operam extensivamente sobre os dados da *Data Class* e mover esses comportamentos para dentro da própria *Data Class*. **Extrair Método** (*Extract Method*) pode ser usado no código cliente para isolar esses comportamentos antes de movê-los.
  - Após enriquecer a classe com comportamento, pode ser possível restringir o acesso direto aos dados, por exemplo, usando **Remover Método Setter** (*Remove Setting Method*) para campos que não devem ser alterados após a criação, ou **Ocultar Método** (*Hide Method*) para getters que expõem detalhes internos desnecessariamente.
- **Impacto da Refatoração:** Melhora a coesão, pois os dados e as operações sobre esses dados são agrupados em um único local. Isso melhora a compreensão e a organização do código. Também ajuda a identificar e eliminar a duplicação de lógica que poderia estar espalhada no código cliente.

4.4. Código Morto (*Dead Code*) (Revisitado/Expandido)

Definição, Sintomas e Cenários Comuns:
Código Morto refere-se a variáveis, parâmetros, campos, métodos ou classes inteiras que não são mais utilizados ou alcançáveis dentro do programa.3 Isso inclui trechos de código em condicionais complexas que nunca são executados devido a condições que sempre avaliam da mesma forma ou erros lógicos.4 Sintomas comuns incluem segmentos de código que nenhuma ferramenta de cobertura de teste consegue atingir, variáveis declaradas e inicializadas mas nunca lidas, ou métodos privados que não são chamados por nenhum outro método na mesma classe.

Refatoração: Identificando e Eliminando Código Morto:
A maneira mais rápida de encontrar código morto é utilizando boas IDEs (Ambientes de Desenvolvimento Integrado) como IntelliJ IDEA ou Eclipse, que possuem análises estáticas embutidas, ou ferramentas de análise estática dedicadas como PMD e SpotBugs (sucessor do FindBugs). Estas ferramentas são altamente eficazes na detecção automática de vários tipos de código não utilizado.
As principais técnicas de refatoração incluem:

- **Excluir Código/Arquivos Não Utilizados:** O tratamento mais direto é simplesmente remover o código ou arquivos que não são mais necessários.[3] Os sistemas de controle de versão servem como uma rede de segurança, permitindo a recuperação caso algo seja removido por engano.
- **Embutir Classe (*Inline Class*) ou Agrupar Hierarquia (*Collapse Hierarchy*):** Se uma classe inteira se torna desnecessária, especialmente se for uma subclasse ou superclasse sem um propósito distinto claro, estas técnicas podem ser aplicadas para fundir a sua funcionalidade noutra classe e remover a classe redundante.
- **Remover Parâmetro (*Remove Parameter*):** Se um parâmetro de um método não é mais utilizado dentro do corpo do método, ele deve ser removido da assinatura do método.

Ferramentas de cobertura de código, como JaCoCo, podem ajudar indiretamente a identificar código morto, destacando seções do código que não são executadas durante os testes. Embora baixa cobertura não signifique necessariamente código morto (pode indicar testes insuficientes), código consistentemente não coberto merece investigação.

Exemplo em Java: Ilustrando Código Morto e Sua Remoção:

- **Antes:**

```
Java
public class ReportGenerator {
    private String reportType; // Campo não utilizado
    private boolean isGenerated = false; // Potencialmente morto se não usado de forma significativa

    public ReportGenerator(String reportType) {
        // this.reportType = reportType; // Comentado ou removido, tornando o campo reportType morto
    }

    // O parâmetro 'includeHeader' pode ser código morto se não for usado
```

```
    public void generateReport(String data, boolean includeHeader) {
        System.out.println("Gerando relatório com dados: " + data);
        // Lógica que não utiliza includeHeader
        if (isGenerated) { // Este branch pode ser morto se isGenerated for sempre false
            // performAdditionalSteps();
        }
        finalizeReport();
    }

    // Método morto se nunca for chamado
    private void performAdditionalSteps() {
        System.out.println("Executando passos adicionais...");
    }

    public void finalizeReport() {
        System.out.println("Relatório finalizado.");
        // isGenerated = true; // Se este fosse o único local para definir como true
    }

    // Método morto se não for chamado de fora e não estiver sobrescrevendo um método de superclasse
    public void oldFeatureCalculation(int x) {
        if (x > 10) {
            //...
        } else {
            // Este 'else' pode ser código morto se 'x' for sempre > 10 em todos os locais de chamada
        }
    }
}
```

- **Depois:**

```
Java
public class ReportGenerator {
    // private String reportType; // Removido
    // private boolean isGenerated = false; // Removido ou refatorado se a lógica mudou

    public ReportGenerator() {
        // Construtor simplificado
    }
```

```
    public void generateReport(String data) { // 'includeHeader' removido
        System.out.println("Gerando relatório com dados: " + data);
        finalizeReport();
    }

    // private void performAdditionalSteps(); // Removido

    public void finalizeReport() {
        System.out.println("Relatório finalizado.");
    }

    // public void oldFeatureCalculation(int x) // Removido
}
```

4.5. Classe Preguiçosa (*Lazy Class*)

Definição: Quando uma Classe Não Justifica Sua Existência:
Uma Classe Preguiçosa é aquela que realiza tão pouca funcionalidade útil que não justifica a sua própria existência como uma entidade separada no sistema. Manter e compreender classes sempre acarreta custos de tempo e recursos; portanto, uma classe deve "pagar o seu sustento" com a funcionalidade que oferece. Frequentemente, estas classes demonstram uma "falta distinta de diversidade" ou comportamento insuficiente.

Técnicas de Refatoração:

- **Embutir Classe (*Inline Class*):** Se uma classe é quase inútil, as suas responsabilidades e funcionalidades são movidas para outra classe que a utiliza, e a classe preguiçosa é então removida.[8] O processo geralmente envolve criar os campos e métodos públicos da classe doadora na classe receptora, substituir todas as referências à classe doadora por referências aos novos membros na classe receptora, testar, e então mover gradualmente toda a funcionalidade até que a classe original esteja vazia e possa ser deletada.[10]
- **Agrupar Hierarquia (*Collapse Hierarchy*):** Se uma classe preguiçosa é uma subclasse com muito poucas funções ou comportamentos distintos em comparação com a sua classe pai, ela pode ser fundida na sua superclasse.[8] Da mesma forma, uma superclasse subutilizada pode ser fundida numa das suas subclasses se fizer sentido no contexto do design.

Exemplo em Java: Identificando uma Classe Preguiçosa e Refatorando com

Embutir Classe:

- **Antes:**

```java
// Classe Preguiçosa: Faz muito pouco por conta própria
class BasicAddressFormatter {
    public String formatAddress(String street, String city, String zip) {
        // Talvez costumasse fazer mais, mas agora apenas concatena
        return street + ", " + city + ", " + zip;
    }
}

class Customer {
    private String name;
    private String street;
    private String city;
    private String zip;
    private BasicAddressFormatter formatter = new BasicAddressFormatter(); // Usa a classe preguiçosa

    public Customer(String name, String street, String city, String zip) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.zip = zip;
    }

    public String getFormattedAddress() {
        return formatter.formatAddress(street, city, zip);
    }

    public String getName() {
        return name;
    }
}
```

- **Depois (Embutir Classe):**

```java
class Customer {
    private String name;
```

```
    private String street;
    private String city;
    private String zip;
    // BasicAddressFormatter é removida

    public Customer(String name, String street, String city, String zip) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.zip = zip;
    }

    public String getFormattedAddress() {
        // A funcionalidade de BasicAddressFormatter é embutida
        return street + ", " + city + ", " + zip;
    }

    public String getName() {
        return name;
    }
}
```

Benefícios e Considerações (Quando Ignorar):
Os benefícios de tratar uma Classe Preguiçosa incluem a redução do tamanho do código, manutenção mais fácil e uma estrutura de classes mais simples.8 Eliminar classes desnecessárias libera "largura de banda na sua cabeça".10 No entanto, há situações em que uma Classe Preguiçosa pode ser ignorada. Por vezes, ela é criada intencionalmente para delinear planos para desenvolvimento futuro ou para manter a clareza para um papel específico, ainda que pequeno.8 Um equilíbrio entre simplicidade e clareza é fundamental.

4.6 Generalidade Especulativa (*Speculative Generality*)

Definição: Superengenharia para Requisitos Futuros Desnecessários (YAGNI):
Este smell ocorre quando o código inclui "ganchos" (hooks), casos especiais, classes, métodos ou parâmetros criados para antecipar necessidades futuras que não se materializaram e podem nunca se materializar.11 É o ato de construir algo "apenas por via das dúvidas", o que representa uma violação direta do princípio YAGNI ("You Ain't Gonna Need It").

Técnicas de Refatoração:

- **Agrupar Hierarquia (*Collapse Hierarchy*):** Para classes/interfaces abstratas não utilizadas ou excessivamente abstratas que faziam parte do design especulativo. Esta técnica envolve fundir uma superclasse e uma subclasse quando se tornaram praticamente idênticas ou quando uma delas adiciona pouco valor.
- **Embutir Classe (*Inline Class*) / Embutir Método (*Inline Method*):** Para delegações desnecessárias ou métodos não utilizados criados especulativamente.
- **Remover Parâmetro (*Remove Parameter*):** Para parâmetros adicionados a métodos para uso futuro, mas nunca utilizados.
- **Remover Código Morto / Excluir Campos Não Utilizados:** Se as funcionalidades especulativas (classes, métodos, campos) estiverem inteiramente não utilizadas.Se apenas casos de teste os utilizam, os testes devem ser excluídos primeiro, e depois o código.

Exemplo em Java: Generalidade Especulativa e Simplificação (ex: usando Agrupar Hierarquia ou simplificando):

- **Antes:**

```
Java
// Classe abstrata especulativamente geral
abstract class DataExporter {
    // Talvez nem todos os exportadores precisem disto
    protected abstract void connectToDataSource();
    public abstract String exportData(Object data);
    // Talvez o logging não seja sempre necessário ou seja feito de forma diferente
    protected abstract void logExport(String dataId);

    // Um método "gancho" que pode nunca ser usado por muitas subclasses
    protected void postProcessExport(String dataId) {
        // Implementação padrão vazia, antecipando que alguns exportadores possam precisar dela
    }
}

// Classe concreta que não usa todas as funcionalidades especulativas
class SimpleStringExporter extends DataExporter {
    @Override
    protected void connectToDataSource() {
        // Este exportador não precisa de uma conexão específica à fonte de dados
        System.out.println("Nenhuma conexão específica à fonte de dados necessária para
```

```
SimpleStringExporter.");
    }

    @Override
    public String exportData(Object data) {
        return "Exportado: " + data.toString();
    }

    @Override
    protected void logExport(String dataId) {
        // SimpleStringExporter pode não registar desta forma específica
        System.out.println("Log simples para: " + dataId);
    }

    // Não sobrescreve nem usa postProcessExport
}

// Outra classe que pode ser muito semelhante ou também não usar todos os "ganchos"
// class AdvancedStringExporter extends DataExporter {... }
```

- **Depois (Agrupar Hierarquia se DataExporter não estiver a fornecer muito valor ou outros exportadores forem semelhantes e simples, ou simplificar DataExporter removendo "ganchos" não utilizados):**

```
Java
// Opção 1: Agrupar Hierarquia (se DataExporter adicionar pouco valor e SimpleStringExporter for o tipo principal/único)
// Ou, mais provável para Generalidade Especulativa, simplificar a classe abstrata ou removê-la se não for necessária.
// Vamos assumir que simplificamos removendo métodos especulativos.

interface MinimalDataExporter { // Ou uma classe abstrata mais simples
    String exportData(Object data);
}

class SimpleStringExporter implements MinimalDataExporter {
    // Sem connectToDataSource se não for geralmente necessário
    // Sem logExport se for específico para certos tipos ou tratado de forma diferente
    // Sem postProcessExport

    @Override
```

```
    public String exportData(Object data) {
        // Realizar a exportação necessária
        String exportedString = "Exportado: " + data.toString();
        // Lidar com qualquer logging específico ou pós-processamento diretamente, se necessário
        System.out.println("A registar exportação para dados: " + data.toString());
        return exportedString;
    }
}
```

Benefícios de Abordar a Generalidade Especulativa:
O resultado é um código mais enxuto e simples, que é mais fácil de suportar e entender.14 Evita-se a superengenharia.
Quando Ignorar:
Ao desenvolver um framework ou biblioteca, alguma generalidade pode ser necessária para os utilizadores, mesmo que não seja usada pelo próprio framework.15 Antes de excluir elementos, é importante garantir que não são utilizados exclusivamente por testes unitários para acesso interno ou ações especiais relacionadas a testes.12
A tabela seguinte resume os *code smells* dispensáveis discutidos:

5. Acopladores: Smells Relacionados a Dependências Excessivas Entre Classes

Os *code smells* da categoria "Acopladores" (*Couplers*) indicam um acoplamento excessivo entre classes ou módulos, ou problemas que surgem quando o acoplamento é substituído por delegação excessiva. O acoplamento forte significa que os componentes dependem fortemente dos detalhes internos uns dos outros, tornando-os difíceis de alterar, substituir ou testar isoladamente. Isso pode levar a efeitos cascata, onde uma alteração num módulo quebra outros. O objetivo é alcançar um baixo acoplamento e alta coesão, frequentemente promovidos pelos princípios SOLID e por padrões de design.

5.1 Inveja de Funcionalidades (*Feature Envy*)

Definição: Um Método Mais Interessado em Outra Classe:
Um método exibe Feature Envy quando acede aos dados ou métodos de outra classe com mais frequência do que aos seus próprios. O método parece "invejar" as funcionalidades da outra classe e pode estar no local errado dentro da estrutura do código.
Este smell ocorre frequentemente após os dados serem movidos para uma classe de dados, mas as operações sobre esses dados são deixadas para trás.21 Pode também surgir de um mal-entendido sobre onde a responsabilidade deve residir. Os problemas incluem a violação

do encapsulamento 24, o aumento do acoplamento entre classes 22, a redução da coesão da classe de origem 24 e a complicação dos testes.24 É um indicador de um design de classe pobre.

Técnica de Refatoração: Mover Método (e potencialmente Extrair Método primeiro):
A solução primária é Mover Método (Move Method): relocar o método "invejoso" para a classe da qual ele está "invejando" as funcionalidades. Isso coloca o comportamento mais próximo dos dados que ele opera. Se apenas parte de um método exibe Feature Envy, a técnica Extrair Método (Extract Method) deve ser usada primeiro para isolar a parte invejosa, e então esse método extraído pode ser movido. Ferramentas como JDeodorant podem detectar Feature Envy e sugerir refatorações de Mover Método.

Exemplo em Java: Detectando Feature Envy e Aplicando Mover Método:

- **Antes:**

```
Java
class Customer {
    private String name;
    private Address address; // Address é outra classe

    public Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String getName() { return name; }
    public Address getAddress() { return address; }
}

class Address {
    private String street;
    private String city;
    private String zipCode;

    public Address(String street, String city, String zipCode) {
        this.street = street;
        this.city = city;
        this.zipCode = zipCode;
    }

    public String getStreet() { return street; }
```

```
    public String getCity() { return city; }
    public String getZipCode() { return zipCode; }

    // Este método pode pertencer aqui se for uma utilidade geral de endereço
    public String getFullAddressLine() {
        return street + ", " + city + " " + zipCode;
    }
}

// Classe exibindo Feature Envy
class OrderPrinter {
    public void printOrderShippingLabel(Order order) {
        Customer customer = order.getCustomer();
        Address custAddress = customer.getAddress(); // Obtendo objeto Address

        // Este método é "invejoso" dos dados de Address
        // Faz múltiplas chamadas aos getters de Address
        String label = customer.getName() + "\n" +
                       custAddress.getStreet() + "\n" +
                       custAddress.getCity() + ", " + custAddress.getZipCode();
        System.out.println(label);
    }
}

class Order { // Classe auxiliar para o exemplo
    private Customer customer;
    public Order(Customer customer) { this.customer = customer; }
    public Customer getCustomer() { return customer; }
}
```

- **Depois (Mover Método - conceitualmente, a lógica de formatação da etiqueta move-se para Address ou um método é adicionado a Address):**

```
Java
class Customer { /*... mesmo que antes... */
    private String name;
    private Address address;

    public Customer(String name, Address address) {
        this.name = name;
```

```
        this.address = address;
    }
    public String getName() { return name; }
    public Address getAddress() { return address; }
}

class Address {
    private String street;
    private String city;
    private String zipCode;

    public Address(String street, String city, String zipCode) {
        this.street = street;
        this.city = city;
        this.zipCode = zipCode;
    }

    public String getStreet() { return street; }
    public String getCity() { return city; }
    public String getZipCode() { return zipCode; }

    // Método movido para aqui (ou um novo criado)
    public String getShippingLabelLine() {
        return street + "\n" + city + ", " + zipCode;
    }
}

class OrderPrinter {
    public void printOrderShippingLabel(Order order) {
        Customer customer = order.getCustomer();
        // Agora chama o método em Address, não acedendo mais diretamente aos seus campos
        String label = customer.getName() + "\n" +
                       customer.getAddress().getShippingLabelLine();
        System.out.println(label);
    }
}
class Order { /*... mesmo que antes... */
    private Customer customer;
    public Order(Customer customer) { this.customer = customer; }
```

```
    public Customer getCustomer() { return customer; }
}
```

Benefícios de Resolver Feature Envy:
A resolução deste smell leva a uma menor duplicação de código (se o tratamento de dados for centralizado), melhor organização do código (métodos para tratar dados ficam próximos aos dados reais) e acoplamento reduzido.
Quando Ignorar:
Por vezes, o comportamento é intencionalmente mantido separado dos dados, por exemplo, nos padrões Strategy ou Visitor, onde a flexibilidade de trocar algoritmos ou adicionar operações a uma hierarquia de classes sem modificá-la é um objetivo de design.

5.2. Intimidade Inapropriada (*Inappropriate Intimacy*)

Definição: Classes que Sabem Demais Sobre os Detalhes Internos Umas das Outras:
Este smell ocorre quando uma classe utiliza excessivamente os campos ou métodos internos de outra classe, ou quando duas classes estão fortemente acopladas através de comunicação bidirecional. Boas classes devem saber o mínimo possível umas sobre as outras para facilitar a manutenção e reutilização. O termo "Negociação Interna" (Insider Trading) é por vezes usado como sinónimo.

Razões para Emergência e Problemas Causados:
Pode surgir de uma falta de limites claros ou de responsabilidades bem definidas entre as classes. Também pode ser um efeito colateral da tentativa de partilhar dados ou funcionalidades de forma demasiado direta. Os problemas incluem alto acoplamento, tornando o sistema frágil (alterações numa classe podem facilmente quebrar a outra), difícil de manter e mais complicado para reutilizar componentes independentemente.

Técnicas de Refatoração:

- **Mover Método (*Move Method*) / Mover Campo (*Move Field*):** Mover as partes (campos ou métodos) que estão a ser acedidas de forma inapropriada para a classe que as utiliza primariamente, ou para uma terceira classe comum, se isso fizer sentido. Isto aplica-se se a classe original não precisar verdadeiramente dessas partes.
- **Extrair Classe (*Extract Class*) e Esconder Delegado (*Hide Delegate*):** Se a intimidade se deve a uma necessidade legítima de colaboração sobre um subconjunto de responsabilidades, Extrair Classe pode criar uma nova classe para essas responsabilidades partilhadas, e Esconder Delegado pode gerir a interação, tornando a relação mais "oficial" e controlada.
- **Mudar Associação Bidirecional para Unidirecional (*Change Bidirectional*

***Association to Unidirectional*):** Se as classes são mutuamente interdependentes, tentar quebrar o ciclo tornando a associação unidirecional.

- **Substituir Delegação por Herança (*Replace Delegation with Inheritance*) (ou vice-versa):** Se a intimidade é entre uma subclasse e uma superclasse, estas podem ser opções dependendo da natureza da relação.
- Mover métodos para evitar intimidade inapropriada é uma abordagem geral sugerida.

Exemplo em Java: Intimidade Inapropriada e Refatoração para Melhor Encapsulamento :

- **Antes:**

```
Java
import java.util.Date;
import java.util.List;

// Assumir que OrderRepository está definido noutro local
interface OrderRepository { List<Order> getAllOrders(); }

class Order {
    public Date deliveryDate; // Campo público - acesso direto é uma forma de intimidade
    private boolean processed = false;

    public Order(Date deliveryDate) {
        this.deliveryDate = deliveryDate;
    }

    public Date getDeliveryDate() { // Getter é aceitável
        return deliveryDate;
    }

    // Método que idealmente deveria ser usado por OrderService,
    // mas OrderService pode estar a fazê-lo diretamente
    public void markAsProcessed() {
        this.processed = true;
    }
    public boolean isProcessed() { return processed; }
}

class OrderService {
```

```
    private OrderRepository orderRepository;

    public OrderService(OrderRepository repo) {
        this.orderRepository = repo;
    }

    // Intimidade Inapropriada: OrderService sabe demais sobre a lógica de deliveryDate de Order
    public int calculateLateOrders() {
        List<Order> orders = orderRepository.getAllOrders();
        int lateOrders = 0;
        for (Order order : orders) {
            // Comparação direta de datas, OrderService é íntimo com a lógica de datas de Order
            if (order.getDeliveryDate().before(new Date())) {
                lateOrders++;
            }
        }
        return lateOrders;
    }

    // Outro exemplo de intimidade: OrderService manipula diretamente o estado de Order
    public void processOrder(Order order) {
        //... alguma lógica de processamento...
        // Alterar diretamente o estado interno de Order - demasiado íntimo
        // order.processed = true; // Se 'processed' fosse público, ou usando um setter demasiado baixo nível
        order.markAsProcessed(); // Isto é melhor, mas a decisão de o chamar pode ser demasiado íntima
                                 // se OrderService ditar demasiado do ciclo de vida de Order.
        System.out.println("Pedido processado.");
    }
}
```

- **Depois:**
  Java
  ```
  import java.util.Date;
  import java.util.List;

  interface OrderRepository { List<Order> getAllOrders(); }
  ```

```
class Order {
    private Date deliveryDate; // Encapsulado
    private boolean processed = false;

    public Order(Date deliveryDate) {
        this.deliveryDate = deliveryDate;
    }

    // A responsabilidade de determinar o atraso está agora dentro de Order
    public boolean isLate() {
        return deliveryDate.before(new Date());
    }

    // Order gere as suas próprias alterações de estado
    public void process() {
        //... lógica interna para o que significa um Pedido ser processado...
        this.processed = true;
        System.out.println("Pedido foi processado internamente.");
    }
    public boolean isProcessed() { return processed; }
    public Date getDeliveryDate() { return deliveryDate; } // Getter é aceitável para acesso de leitura
}

class OrderService {
    private OrderRepository orderRepository;

    public OrderService(OrderRepository repo) {
        this.orderRepository = repo;
    }

    public int countAllLateOrders() { // Renomeado para clareza
        List<Order> orders = orderRepository.getAllOrders();
        int lateOrders = 0;
        for (Order order : orders) {
            // OrderService agora pergunta a Order se está atrasado
            if (order.isLate()) {
                lateOrders++;
            }
```

```
        }
        return lateOrders;
    }

    public void processAnOrder(Order order) {
        // OrderService diz ao Pedido para se processar
        order.process();
        // OrderService pode fazer outras coisas relacionadas com o processamento de um pedido
        // que são da sua própria responsabilidade, e.g., notificar outros sistemas.
        System.out.println("OrderService completou a sua parte do processamento do pedido.");
    }
}
```

Benefícios de Reduzir a Intimidade Inapropriada:
A redução deste smell facilita a manutenção e reutilização, pois as classes tornam-se mais independentes. Melhora também o encapsulamento.

5.3. Cadeias de Mensagens (*Message Chains*)

Definição: Longas Sequências de Chamadas de Métodos (Violando a Lei de Deméter):
Uma Cadeia de Mensagens ocorre quando o código envolve uma sequência de chamadas como objetoA.getObjetoB().getObjetoC().fazerAlgumaCoisa(). O cliente solicita um objeto, que por sua vez solicita outro, e assim por diante. Isso frequentemente viola a Lei de Deméter (Princípio do Menor Conhecimento), que afirma que um objeto deve chamar apenas métodos dos seus "amigos" diretos.
Razões para Emergência e Problemas Causados (Fragilidade, Acoplamento Forte à Navegação):
Podem surgir da exposição excessiva de estruturas internas de objetos. O código do cliente torna-se dependente do caminho de navegação ao longo da estrutura de classes. Os problemas incluem tornar o código frágil; qualquer alteração nas relações intermediárias da cadeia exige a modificação do cliente. Acopla fortemente o cliente à estrutura interna de múltiplos objetos e torna o código difícil de ler, entender e manter.

Técnicas de Refatoração:

- **Esconder Delegado (*Hide Delegate*):** A técnica primária. Cria-se um novo método no objeto no início da cadeia (ou num objeto intermediário) que encapsula a cadeia, escondendo a delegação do cliente. O cliente então chama este único método.
- **Extrair Método (*Extract Method*) e Mover Método (*Move Method*):** Considerar por que o objeto final está a ser usado. Pode fazer sentido extrair a

funcionalidade realizada pelo final da cadeia para o seu próprio método e, em seguida, mover este método para uma classe anterior na cadeia, ou mesmo para a primeira classe.
- **Introduzir Variáveis Intermediárias (*Introduce Intermediate Variables*):** Pode decompor a cadeia para melhorar a legibilidade, embora isso não resolva fundamentalmente o problema do acoplamento.

Exemplo em Java: Uma Cadeia de Mensagens e Sua Refatoração usando Esconder Delegado:

- **Antes:**

```
Java
// Assumir que estas classes existem e têm os respetivos métodos getter
class SaveItem {
    private boolean enabled;
    public void setEnabled(boolean enabled) { this.enabled = enabled;
System.out.println("SaveItem enabled: " + enabled); }
    public boolean isEnabled() { return enabled; }
}
class Customizer {
    private SaveItem saveItem = new SaveItem();
    public SaveItem getSaveItem() { return saveItem; }
}
class DockablePanel {
    private Customizer customizer = new Customizer();
    public Customizer getCustomizer() { return customizer; }
}
class Modelisable {
    private DockablePanel dockablePanel = new DockablePanel();
    public DockablePanel getDockablePanel() { return dockablePanel; }
}
class MasterControl {
    private Modelisable modelisable = new Modelisable();
    public Modelisable getModelisable() { return modelisable; }

    public void disableSaving() {
        // Cadeia de Mensagens: Longa sequência de getters

this.getModelisable().getDockablePanel().getCustomizer().getSaveItem().setEnabl
ed(false);
```

```
    }
     public void enableSaving() {

this.getModelisable().getDockablePanel().getCustomizer().getSaveItem().setEnabl
ed(true);
    }
}
```

- **Depois (Esconder Delegado em MasterControl ou Modelisable):**

```
Java
// SaveItem, Customizer, DockablePanel permanecem os mesmos para este exemplo específico de
refatoração.
// Modelisable pode ser um bom local para esconder parte da cadeia.

class Modelisable {
    private DockablePanel dockablePanel = new DockablePanel();
    // getDockablePanel() pode tornar-se privado ou package-private
    // public DockablePanel getDockablePanel() { return dockablePanel; }

    // Novo método para esconder a delegação para ativar/desativar a funcionalidade de gravação
    public void setSaveFunctionalityEnabled(boolean enabled) {
        this.dockablePanel.getCustomizer().getSaveItem().setEnabled(enabled);
    }
}

class MasterControl {
    private Modelisable modelisable = new Modelisable();
    // getModelisable() pode não ser mais necessário publicamente se todas as interações
    // passarem por novos métodos.

    public void disableSaving() {
        // O cliente (MasterControl) agora chama um único método em Modelisable
        modelisable.setSaveFunctionalityEnabled(false);
    }

    public void enableSaving() {
        modelisable.setSaveFunctionalityEnabled(true);
    }
}
```

Benefícios e Considerações (Evitando o smell de Middle Man):
Os benefícios incluem a redução de dependências entre classes na cadeia e a diminuição de código cliente inchado, além de melhorar a manutenibilidade. No entanto, uma ocultação de delegação excessivamente agressiva pode levar ao smell de Middle Man, onde uma classe acaba com muitos métodos que apenas delegam, tornando difícil ver onde a funcionalidade realmente ocorre. É necessário um equilíbrio.

3.5. Homem do Meio (*Middle Man*)

Definição: Uma Classe que Delega Primariamente o Trabalho:
Uma classe Middle Man é aquela onde a maioria dos seus métodos simplesmente delega chamadas para métodos em outra classe. Se uma classe realiza apenas uma ação – delegar – a sua existência é questionável.
Razões para Emergência (ex: Remoção Excessivamente Agressiva de Cadeias de Mensagens) e Problemas Causados:
Pode ser o resultado de uma aplicação excessivamente zelosa de Esconder Delegado ao remover Cadeias de Mensagens. Pode também ocorrer se o trabalho útil de uma classe for gradualmente movido para outras classes, deixando uma "casca vazia". Os problemas incluem a adição de complexidade desnecessária e uma camada extra de indireção, aumentando o tamanho do código sem adicionar valor significativo. Pode ser cansativo se cada nova funcionalidade no delegado exigir um novo método de delegação no servidor.

Técnica de Refatoração: Remover Homem do Meio (Remove Middle Man):
O cliente deve chamar o objeto delegado diretamente. Isso envolve:

1. Criar um método *getter* na classe servidora (*Middle Man*) para aceder ao objeto delegado.
2. Substituir as chamadas do cliente aos métodos de delegação do *Middle Man* por chamadas diretas aos métodos do delegado, obtidos através do novo *getter*. Se a classe não fizer mais nada, deve-se considerar se ela é realmente necessária.

Exemplo em Java: Identificando um Middle Man e Aplicando Remover Middle Man:

- **Antes:**

```
Java
class RealWorker {
    public void performTask() {
        System.out.println("RealWorker: Executando a tarefa real.");
    }
}
```

```
// Classe MiddleMan
class Manager {
    private RealWorker worker = new RealWorker();

    // Método de delegação
    public void assignTask() {
        worker.performTask(); // Simplesmente delega para RealWorker
    }

    // Potencialmente muitos outros métodos que apenas delegam para RealWorker
    // public String getWorkerStatus() { return worker.getStatus(); }
}

class Client {
    public void initiateWork() {
        Manager manager = new Manager();
        manager.assignTask(); // Cliente fala com o Manager
    }
}
```

- **Depois (Remover Middle Man):**

```
Java
class RealWorker {
    public void performTask() {
        System.out.println("RealWorker: Executando a tarefa real.");
    }
}

class Manager { // Manager ainda pode existir se tiver outras responsabilidades não delegatórias
    private RealWorker worker = new RealWorker();

    // Getter para o delegado
    public RealWorker getDelegateWorker() {
        return worker;
    }
    // assignTask() é removido, ou o papel do Manager muda.
    // Se Manager tivesse outros papéis, eles permaneceriam.
    // Se assignTask fosse o seu único papel, e agora o Cliente obtém o worker,
    // Manager poderia tornar-se uma Classe Preguiçosa ou ser removido.
```

```
}

class Client {
    public void initiateWork() {
        Manager manager = new Manager(); // Cliente ainda pode precisar do Manager por outras razões
        RealWorker actualPerformer = manager.getDelegateWorker(); // Cliente obtém o trabalhador real
        actualPerformer.performTask(); // Cliente fala diretamente com RealWorker
    }
}

// Alternativa se o único propósito do Manager fosse a delegação:
// class Client {
//     public void initiateWork() {
//         RealWorker worker = new RealWorker(); // Cliente cria/obtém RealWorker diretamente
//         worker.performTask();
//     }
// }
// Neste caso, a classe Manager poderia ser removida completamente.
```

Benefícios e Quando Ignorar (Middle Men Intencionais em Padrões de Design):
Os benefícios incluem código menos volumoso e complexidade reduzida.34 No entanto, não se deve remover Middle Men criados por razões específicas e válidas 34:

- Para evitar dependências entre classes (atuando como uma camada de desacoplamento deliberada).
- Quando fazem parte de padrões de design como Proxy, Decorator ou Facade, que utilizam intencionalmente uma estrutura de "homem do meio".