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.