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: