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.