REFATORAÇÃO

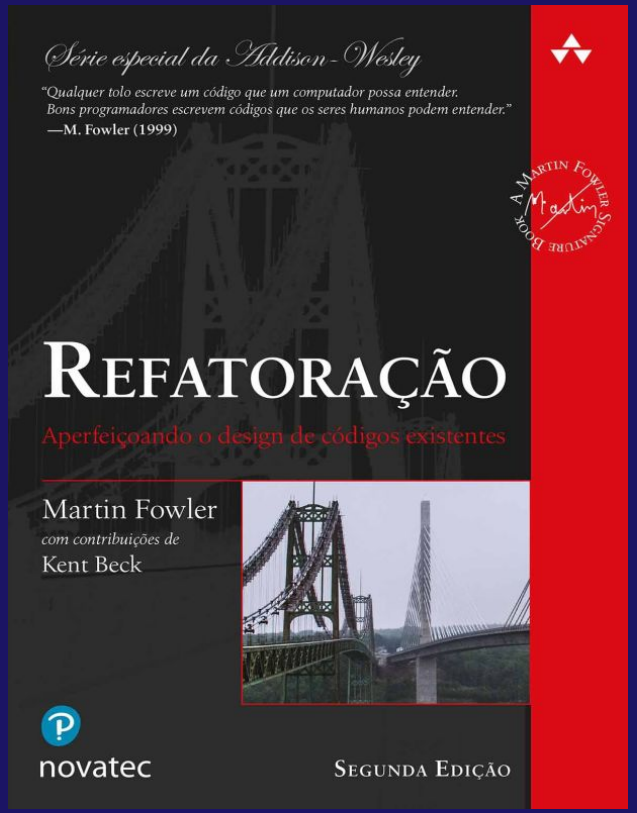
A arte de salvar sistemas legados

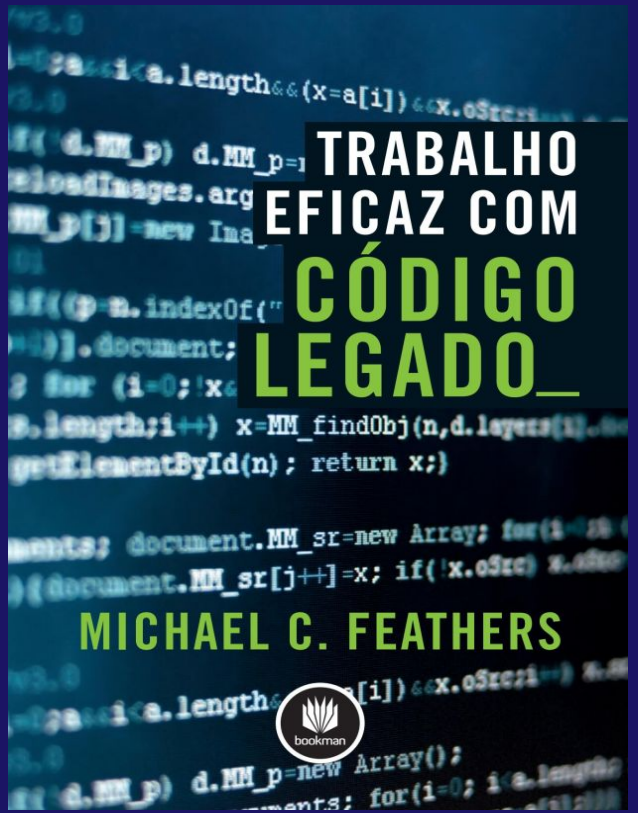
Olá

Eu sou Willian Brito

- Analista de Sistemas na **Aiko**.
- **Formado** em Análise e Desenvolvimento de Sistemas.
- Pós Graduado em **Segurança Cibernética**.
- Certificação **SYCP** (Solyd Certified Pentester) v2018.

ESTE CONTEÚDO FOI BASEADO NESTAS OBRAS





“Refatoração é o processo de melhorar o design e a estrutura interna do código sem alterar seu comportamento externo.”

—MARTIN FOWLER

TÓPICOS ABORDADOS

01
Introdução

02
Code Smells

03
Catálogo de Técnicas

04
Testes Automatizados

01

INTRODUÇÃO

INTRODUÇÃO

Refatoração é o processo de melhorar o design e a estrutura interna do código sem alterar seu comportamento visível. Para mim, é uma prática essencial porque permite manter o código limpo, organizado e pronto para crescer junto com o sistema.

Quando estamos desenvolvendo, é comum que a pressão por entregas rápidas acabe gerando um código mais confuso ou fragmentado, e é aí que entra a refatoração como uma solução para dar clareza e coesão ao projeto.

O objetivo principal da refatoração é deixar o código mais fácil de entender e mais simples de manter. Ao melhorar a organização, reduzimos a chance de introduzir erros e facilitamos o processo de adicionar novas funcionalidades.

INTRODUÇÃO

Basicamente, quando refatoro um trecho de código, estou reorganizando e ajustando a estrutura interna para eliminar complexidades desnecessárias e torná-lo mais direto. Isso pode envolver, por exemplo, extrair partes de um método grande em funções menores, renomear variáveis para torná-las mais descritivas ou até mover métodos para classes onde eles façam mais sentido.

Outra razão pela qual valorizo a refatoração é que ela ajuda a eliminar o que chamamos de **"code smells"** que são sinais de que o código pode ser melhorado. Esses "cheiros" indicam problemas de design, como duplicação de código, métodos longos e classes sobrecarregadas. Quando identifico esses indícios, vejo como uma oportunidade de refatorar, reduzindo complexidade e facilitando a manutenção.

INTRODUÇÃO

No caso de código legado, a refatoração se torna ainda mais essencial. Para **Michael Feathers "Código legado é simplesmente código sem testes"**, ou seja, na visão do autor código legado é aquele que não possui uma base de testes automatizados, o que dificulta a compreensão e a segurança ao realizar alterações.

Uma abordagem que considero eficaz é inserir testes gradualmente para entender o comportamento do sistema e garantir que as mudanças não afetem seu funcionamento. Essa base de testes me permite refatorar com mais confiança, dividindo responsabilidades e organizando o código em partes mais controladas e modulares.

INTRODUÇÃO

Em resumo, para mim, a refatoração é mais do que apenas ajustar o código: é uma prática contínua que garante a saúde do sistema, possibilitando que ele evolua sem comprometer sua integridade. É uma estratégia que nos permite combater o acúmulo de dívidas técnicas e manter um padrão de qualidade que vai facilitar tanto o trabalho da equipe quanto a experiência dos usuários.

02

CODE SMELLS

CODE SMELLS

Code smells, ou **"cheiros de código"**, são sinais de que algo no código pode ser melhorado para manter sua qualidade, organização e legibilidade. Estes problemas não são necessariamente erros, mas indicam potenciais pontos de fragilidade que podem dificultar a manutenção e aumentar a complexidade do sistema.

Aqui estão os principais code smells e o que eles geralmente indicam:

1. Duplicação de Código (Duplicated Code): Ocorre quando blocos de código idênticos ou muito semelhantes aparecem em mais de um lugar. Isso aumenta o esforço de manutenção, pois qualquer mudança precisa ser replicada em todos os locais. Geralmente, é melhor consolidar o código repetido em um método ou função única, que possa ser reutilizada.

CODE SMELLS

2. Métodos Longos (Long Method): Métodos extensos são difíceis de entender e testar, e frequentemente fazem mais do que deveriam. Reduzir o tamanho dos métodos, extraindo partes em métodos menores e mais focados, ajuda a manter o código mais legível e modular.

3. Classes Grandes (Large Class): Classes com muitas responsabilidades podem se tornar difíceis de manter e entender. Este code smell sugere que a classe pode ser dividida em outras classes menores, seguindo o princípio de responsabilidade única.

4. Lista de Parâmetros Grande (Long Parameter List): Funções ou métodos que recebem muitos parâmetros se tornam difíceis de entender e podem indicar que esses parâmetros deveriam estar encapsulados em um objeto. Reduzir a quantidade de parâmetros facilita o uso e a compreensão das funções.

CODE SMELLS

5. Comentários Excessivos (Comments): Embora comentários sejam importantes para documentar o código, o uso excessivo pode indicar que o código em si é confuso e precisa de melhorias. Idealmente, o código deve ser autoexplicativo, bons nomes e uma estrutura clara muitas vezes eliminam a necessidade de comentários.

6. Código Duplicado em Diversos Locais (Shotgun Surgery): Ocorre quando uma pequena mudança no sistema requer modificações em vários lugares. Isso dificulta a manutenção e aumenta o risco de erros, já que pode ser fácil esquecer de atualizar algum local. Refatorar para centralizar essa lógica em um único local é geralmente a solução.

7. Inveja de Função (Feature Envy): Acontece quando um método em uma classe utiliza mais dados ou métodos de outra classe do que da sua própria. Isso indica que o método pode estar no lugar errado e poderia ser movido para a classe que mais utiliza.

CODE SMELLS

8. Middle Man (Intermediário): Quando uma classe se limita a passar chamadas para outra classe sem acrescentar nada, ela pode ser uma intermediária desnecessária. Remover ou simplificar esse intermediário pode ajudar a reduzir a complexidade.

9. Classes Preguiçosas (Lazy Class): Classes que contêm pouco código ou têm pouca responsabilidade podem ser um sinal de que não são necessárias. Nesses casos, faz sentido integrá-las a outras classes para simplificar a estrutura.

10. Obsessão por Tipos Primitivos (Primitive Obsession): Usar tipos primitivos (como `int` ou `string`) para representar conceitos complexos, como datas ou informações de contato, pode indicar que seria melhor encapsular esses dados em classes específicas. Isso melhora a clareza e evita bugs relacionados à manipulação direta desses valores.

CODE SMELLS

11. Classe Pai Inapropriada (Refused Bequest): Quando uma classe filha não utiliza ou precisa de parte da funcionalidade herdada da classe pai, isso indica que a hierarquia de classes pode estar incorreta. É possível repensar a estrutura de herança ou introduzir uma classe mais genérica para acomodar as necessidades.

12. Diversas Responsabilidades em Uma Classe (God Object ou God Class): Uma "classe Deus" é uma classe central que faz muitas coisas diferentes e controla diversas partes do sistema. Dividir essa classe em várias classes menores e mais focadas é uma forma de reduzir o acoplamento e melhorar a manutenção.

13. Excesso de Condicionais (Excessive Conditionals): Muitos `ifs`, `elses` e `switches` podem tornar o código mais confuso e difícil de manter. Estruturas complexas de decisão podem ser substituídas por polimorfismo ou strategy pattern, o que simplifica o código e torna o fluxo de decisão mais claro.

CODE SMELLS

14. Valores Mágicos (Magic Numbers): São valores numéricos ou strings diretamente no código sem qualquer explicação. Eles dificultam o entendimento do propósito desses valores. A prática recomendada é substituir esses valores por constantes nomeadas.

15. Complexidade Ciclomática Alta: Refere-se ao número de caminhos independentes em um trecho de código, como funções que têm muitos `if` ou `loop` aninhados. A alta complexidade ciclomática indica que o código é difícil de entender e testar, e talvez precise ser dividido em funções menores e mais simples.

16. Especificidade Excessiva (Speculative Generality): Criar abstrações ou classes que ainda não têm uso real pode gerar confusão e complexidade desnecessária. É melhor evitar a criação de código "no caso de" ser necessário futuramente, focando no que realmente é necessário no momento.

CODE SMELLS

17. Classes com Dados Temporários (Temporary Field): Quando uma classe possui atributos que só são utilizados em alguns cenários, isso indica que esses dados podem não pertencer a ela. Refatorar esses dados para classes específicas reduz a confusão.

18. Inconsistência de Nomes (Inconsistent Naming): Quando nomes de variáveis, métodos ou classes não seguem um padrão, o código se torna mais difícil de ler e entender. Definir e seguir uma convenção de nomes é essencial para a clareza do código.

19. Encapsulamento Fraco (Inappropriate Intimacy): Ocorre quando uma classe conhece muitos detalhes internos de outra, quebrando o princípio de encapsulamento. Minimizar o conhecimento que uma classe tem sobre a outra fortalece a modularidade e reduz o acoplamento.

CODE SMELLS

20. Acoplamento Excessivo (Tight Coupling): Quando as classes estão muito dependentes umas das outras, é difícil modificar uma sem impactar outras. Reduzir o acoplamento, por meio de interfaces ou inversão de dependência, melhora a flexibilidade e facilita a manutenção do sistema.

Cada code smell aponta para uma oportunidade de simplificar, organizar e estruturar melhor o código. Refatorar para eliminar esses sinais mantém a qualidade do software e reduz o custo e o esforço de manutenção a longo prazo.

03

CATÁLOGO DE TÉCNICAS

CATÁLOGO DE TÉCNICAS

O catálogo de técnicas de refatoração é uma coleção de técnicas que organizam e sistematizam as principais práticas de refatoração para problemas específicos de código.

Esses catálogos funcionam como um guia prático, listando várias abordagens para melhorar a qualidade, clareza e manutenibilidade do código, além de explicarem quando e como aplicar cada técnica.

A seguir vamos falar sobre as principais técnicas de refatoração encontrados no catálogo, com exemplos práticos:

CATÁLOGO DE TÉCNICAS

1. Extrair Método (Extract Method): Quando um método contém muito código ou múltiplas responsabilidades, é possível extrair partes desse código em métodos menores, com nomes descritivos. Isso melhora a legibilidade e permite reutilizar o código extraído em outros lugares.

```
// Antes da refatoração
public void ProcessOrder(Order order)
{
    Console.WriteLine("Validating order...");
    Console.WriteLine("Calculating price...");
    Console.WriteLine("Applying discount...");
}

// Após a refatoração
public void ProcessOrder(Order order)
{
    ValidateOrder(order);
    CalculatePrice(order);
    ApplyDiscount(order);
}

private void ValidateOrder(Order order) => Console.WriteLine("Validating order...");
private void CalculatePrice(Order order) => Console.WriteLine("Calculating price...");
private void ApplyDiscount(Order order) => Console.WriteLine("Applying discount...");
```

CATÁLOGO DE TÉCNICAS

2. Método Inline (Inline Method): Essa técnica é útil quando um método não faz nada além de chamar outro método ou encapsular uma lógica muito simples. Nesses casos, remover o método e colocar seu código diretamente onde ele é chamado simplifica a estrutura.

```
// Antes da refatoração
public bool IsEligibleForDiscount(Order order) => HasLoyaltyPoints(order);

private bool HasLoyaltyPoints(Order order) => order.LoyaltyPoints > 100;

// Após a refatoração
public bool IsEligibleForDiscount(Order order) => order.LoyaltyPoints > 100;
```

CATÁLOGO DE TÉCNICAS

3. Renomear Método (Rename Method): Renomear métodos ou variáveis para termos mais descritivos é uma forma de melhorar a legibilidade do código. Bons nomes ajudam os desenvolvedores a entender rapidamente o que cada método ou variável faz, sem a necessidade de comentários extras.

```
// Antes da refatoração
public void CalcSal() { /* cálculo de salário */ }

// Após a refatoração
public void CalculateSalary() { /* cálculo de salário */ }
```

CATÁLOGO DE TÉCNICAS

4. Extrair Variável (Extract Variable): Quando uma expressão complexa é usada repetidamente ou torna o código difícil de ler, ela pode ser extraída para uma variável com um nome descritivo. Isso facilita o entendimento da expressão e pode simplificar o processo de manutenção.

```
// Antes da refatoração
if (order.TotalPrice * 0.08 > 100)
{
    // lógica
}

// Após a refatoração
decimal taxAmount = order.TotalPrice * 0.08;
if (taxAmount > 100)
{
    // lógica
}
```

CATÁLOGO DE TÉCNICAS

5. Substituir Número Mágico por Constante (Replace Magic Number with Symbolic Constant): Substituir números ou strings "mágicos" (valores sem contexto claro) por constantes nomeadas aumenta a clareza. Com nomes descritivos, é mais fácil entender o propósito desses valores.

```
// Antes da refatoração
public void ApplyDiscount() => Price *= 0.9;

// Após a refatoração
private const decimal DiscountRate = 0.9m;
public void ApplyDiscount() => Price *= DiscountRate;
```

CATÁLOGO DE TÉCNICAS

6. Encapsular Campo (Encapsulate Field): Em vez de acessar diretamente os campos de uma classe, é possível encapsulá-los usando métodos `get` e `set`. Isso protege o campo de modificações diretas e permite implementar lógica adicional, como validações.

```
// Antes da refatoração
0 references
public string Name;

// Após a refatoração
0 references
private string name;
0 references
public string Name
{
    get => name;
    set => name = value;
}
```

CATÁLOGO DE TÉCNICAS

7. Substituir Tipo Primitivo por Objeto (Replace Primitive with Object): Quando um tipo primitivo é utilizado para representar uma entidade mais complexa, ele pode ser substituído por uma classe. Por exemplo, em vez de usar uma `string` para representar um número de telefone, criar uma classe específica para esse propósito permite encapsular validações e formatações.

```
// Antes da refatoração
public class Order
{
    public string CustomerName;
}

// Após a refatoração
public class Order
{
    public Customer Customer;
}

public class Customer
{
    public string Name;
}
```

CATÁLOGO DE TÉCNICAS

8. Extrair Classe (Extract Class): Quando uma classe começa a ter responsabilidades demais, parte de suas variáveis e métodos pode ser movida para uma nova classe. Isso ajuda a distribuir as responsabilidades de forma mais equilibrada, facilitando a manutenção.

```
// Antes da refatoração
public class Employee
{
    public string Name;
    public string Address;
    public string PhoneNumber;
}

// Após a refatoração
public class Employee
{
    public string Name;
    public ContactInfo ContactInfo;
}

public class ContactInfo
{
    public string Address;
    public string PhoneNumber;
}
```

CATÁLOGO DE TÉCNICAS

9. Introduzir Objeto de Parâmetro (Introduce Parameter Object): Quando um método possui muitos parâmetros, é possível agrupá-los em uma nova classe ou objeto. Isso simplifica a lista de parâmetros e torna o código mais fácil de ler e modificar.

```
// Antes da refatoração
public void CreateOrder(string customerName, string product, int quantity) { /* lógica */ }

// Após a refatoração
public void CreateOrder(OrderInfo orderInfo) { /* lógica */ }

public class OrderInfo
{
    public string CustomerName;
    public string Product;
    public int Quantity;
}
```

CATÁLOGO DE TÉCNICAS

10. Substituir Herança por Delegação (Replace Inheritance with Delegation): Em alguns casos, a herança pode ser substituída por delegação para reduzir o acoplamento. Ao invés de herdar métodos de uma superclasse, uma classe pode ter um campo que aponta para um objeto e delegar as chamadas a ele.

```
// Antes da refatoração
1 reference
public class ElectricCar : Car { }

// Após a refatoração
1 reference
public class ElectricCar
{
    0 references
    private Car car = new Car();
}
```

CATÁLOGO DE TÉCNICAS

11. Mover Método (Move Method): Se um método em uma classe usa mais dados e métodos de outra classe do que da sua própria, ele pode ser movido para a classe onde faz mais sentido. Isso ajuda a manter a lógica de forma coesa.

```
// Antes da refatoração
public class Order
{
    public decimal GetTotalPrice() { /* cálculo do preço */ }
}

// Após a refatoração
public class Order
{
    public PriceCalculator PriceCalculator = new PriceCalculator();
}

public class PriceCalculator
{
    public decimal GetTotalPrice(Order order) { /* cálculo do preço */ }
}
```

CATÁLOGO DE TÉCNICAS

12. Mover Campo (Move Field): Similar ao mover método, essa técnica permite transferir campos para a classe onde eles são mais utilizados. Isso reduz o acoplamento e melhora a organização.

```
// Antes da refatoração
public class Order
{
    public decimal TaxRate = 0.08m;
}

// Após a refatoração
public class TaxSettings
{
    public decimal TaxRate = 0.08m;
}

public class Order
{
    public TaxSettings TaxSettings;
}
```

CATÁLOGO DE TÉCNICAS

13. Remover Classe Intermediária (Remove Middle Man): Quando uma classe se limita a redirecionar chamadas para outra, ela pode ser eliminada, e as chamadas podem ir diretamente para a classe final, reduzindo a complexidade e o acoplamento.

```
// Antes da refatoração
0 references
public class Manager
{
    1 reference
    private Employee employee;
    0 references
    public string GetEmployeeName() => employee.Name;
}

// Após a refatoração
1 reference
public class Employee
{
    1 reference
    public string Name;
}
```

CATÁLOGO DE TÉCNICAS

14. Introduzir Método de Fabricação (Introduce Factory Method): Quando a criação de um objeto se torna complexa, um método de fábrica pode ser criado para encapsular essa lógica de construção, facilitando a reutilização e o entendimento.

```
// Antes da refatoração
3 references
public class Car
{
    1 reference
    public Car(string model) { }
}

// Após a refatoração
0 references
public class CarFactory
{
    0 references
    public static Car Create(string model) => new Car(model);
}
```

CATÁLOGO DE TÉCNICAS

15. Remover Parâmetro (Remove Parameter): Se um parâmetro não é mais necessário ou seu valor pode ser acessado de outra forma, ele deve ser removido para simplificar a assinatura do método.

```
// Antes da refatoração
public void CalculateDiscount(decimal discountRate = 0.05m) { /* lógica */ }

// Após a refatoração
private const decimal DefaultDiscountRate = 0.05m;
public void CalculateDiscount() { /* lógica usando DefaultDiscountRate */ }
```

CATÁLOGO DE TÉCNICAS

16. Dividir Loop (Split Loop): Quando um único loop realiza múltiplas tarefas, ele pode ser dividido em loops separados para melhorar a clareza e facilitar o entendimento.

```
// Antes da refatoração
foreach (var order in orders)
{
    CalculateTotal(order);
    SendConfirmation(order);
}

// Após a refatoração
foreach (var order in orders)
{
    CalculateTotal(order);
}

foreach (var order in orders)
{
    SendConfirmation(order);
}
```

CATÁLOGO DE TÉCNICAS

17. Consolidar Condições Condicionais (Consolidate Conditional Expression): Se múltiplas expressões condicionais resultam no mesmo código, elas podem ser consolidadas em uma única condição, tornando o código mais conciso e fácil de ler.

```
// Antes da refatoração
if (age > 65 || hasDisability || isVeteran)
{
    ApplyDiscount();
}

// Após a refatoração
if (IsEligibleForDiscount())
{
    ApplyDiscount();
}

private bool IsEligibleForDiscount() => age > 65 || hasDisability || isVeteran;
```

CATÁLOGO DE TÉCNICAS

18. Desacoplar Interface (Extract Interface): Quando múltiplas classes utilizam métodos similares, uma interface pode ser extraída para garantir que todas sigam um padrão. Isso facilita o uso de polimorfismo e reduz o acoplamento.

```
// Antes da refatoração
public class Car
{
    public void Drive() { }
    public void Refuel() { }
}

// Após a refatoração
public interface IVehicle
{
    void Drive();
}

public class Car : IVehicle
{
    public void Drive() { }
    public void Refuel() { }
}
```

CATÁLOGO DE TÉCNICAS

19. Substituir Condicional por Polimorfismo (Replace Conditional with Polymorphism): Condicionais `if-else` ou `switch` que controlam o comportamento com base em tipos específicos podem ser substituídos por polimorfismo, criando uma estrutura mais clara e modular.

```
// Antes da refatoração
public decimal CalculateShippingCost(Order order)
{
    if (order.Type == "Express") return 30;
    else if (order.Type == "Standard") return 10;
    return 20;
}

// Após a refatoração
public abstract class Order
{
    public abstract decimal GetShippingCost();
}

public class ExpressOrder : Order
{
    public override decimal GetShippingCost() => 30;
}

public class StandardOrder : Order
{
    public override decimal GetShippingCost() => 10;
}
```

CATÁLOGO DE TÉCNICAS

20. Remover Classe Preguiçosa (Remove Lazy Class): Se uma classe não justifica sua existência devido ao pouco uso de métodos ou dados, ela pode ser eliminada ou integrada a outra classe para simplificar a estrutura.

```
// Antes da refatoração
0 references
public class Address
{
    0 references
    public string City;
}

// Após a refatoração
0 references
public class Customer
{
    0 references
    public string City;
}
```

CATÁLOGO DE TÉCNICAS

21. Colocar Método (Introduce Local Extension): Para estender o comportamento de uma classe existente sem modificá-la, pode-se criar uma classe que encapsule a original, adicionando métodos específicos.

```
// Antes da refatoração
// Suponha que você não pode modificar a classe `Customer` da biblioteca externa.

0 references
public class CustomerHelper : Customer
{
    0 references
    public string GetFullName() => $"{FirstName} {LastName}";
}
```

CATÁLOGO DE TÉCNICAS

22. Substituir Array por Objeto (Replace Array with Object): Arrays usados para representar entidades com mais de um campo (como `[nome, idade]`) podem ser substituídos por classes com campos nomeados, aumentando a clareza e a segurança.

```
// Antes da refatoração
string[] employeeData = new string[2];
employeeData[0] = "John";
employeeData[1] = "Doe";

// Após a refatoração
1 reference
public class Employee
{
    1 reference
    public string FirstName;
    1 reference
    public string LastName;
}

var employee = new Employee { FirstName = "John", LastName = "Doe" };
```

CATÁLOGO DE TÉCNICAS

Essas técnicas de refatoração visam simplificar a estrutura do código, melhorar sua legibilidade e reduzir o acoplamento. São passos práticos que ajudam a transformar o código gradualmente, sem alterar seu comportamento, mas garantindo que ele permaneça sustentável e fácil de entender a longo prazo.

04

TESTES AUTOMATIZADOS

TESTES AUTOMATIZADOS

Os testes automatizados são uma peça central no processo de refatoração. Quando trabalhamos em refatoração, nosso objetivo é melhorar a estrutura do código sem alterar seu comportamento externo. Os testes automatizados, quando bem escritos e abrangentes, garantem que o código refatorado continue a produzir os mesmos resultados e que nenhuma funcionalidade seja comprometida durante o processo de aprimoramento.

Importância dos Testes no Processo de Refatoração

1. Segurança e Confiabilidade: A principal razão para ter testes automatizados no processo de refatoração é garantir que a refatoração não introduza erros. Mudanças estruturais no código podem ter impactos inesperados, mas com uma suíte de testes sólida, temos uma rede de segurança. Os testes permitem identificar problemas imediatamente e restaurar o código à sua versão anterior caso algo dê errado.

TESTES AUTOMATIZADOS

2. Rapidez no Feedback: A refatoração é muitas vezes um processo iterativo, onde fazemos pequenas mudanças incrementais. Com testes automatizados, obtemos feedback rápido após cada alteração, permitindo identificar e corrigir erros com agilidade, o que torna o processo de refatoração mais produtivo e eficiente.

3. Evolução Contínua: Um código limpo e bem estruturado facilita a manutenção e evolução da aplicação. Quando há uma suíte de testes abrangente, o desenvolvedor se sente mais confiante para fazer melhorias constantes, sabendo que qualquer alteração será verificada quanto à sua integridade.

4. Prevenção de Regressão: A refatoração pode alterar a estrutura interna sem modificar o comportamento externo. Ter testes em funcionamento antes da refatoração evita que "regressões" (erros que reintroduzem problemas anteriores) ocorram, mantendo a funcionalidade intacta.

TESTES AUTOMATIZADOS

Tipos de Testes Importantes no Processo de Refatoração

- **Testes Unitários:** São a base de uma boa cobertura de código e garantem que as funções e métodos individuais estejam funcionando conforme o esperado. Cada classe e método são testados isoladamente, facilitando identificar quebras específicas.
- **Testes de Integração:** Esses testes verificam como diferentes partes do sistema interagem entre si. Durante a refatoração, especialmente ao mover ou extrair classes e métodos, é crucial garantir que a integração entre os componentes continue intacta.
- **Testes de e2e (end to end):** Testam o sistema como um todo, assegurando que o comportamento do aplicativo corresponda às expectativas e requisitos dos usuários. Eles ajudam a identificar se o sistema, após a refatoração, ainda cumpre com as especificações e fluxos desejados.

TESTES AUTOMATIZADOS

Boas Práticas para Testes na Refatoração

1. Escreva Testes Antes de Refatorar: A prática conhecida como "test-first" é uma abordagem essencial. Antes de alterar o código, escreva testes para cobrir o comportamento atual, garantindo que, ao final, o sistema funcione da mesma maneira.

2. Refatore em Pequenas Partes: Para facilitar o processo de teste, faça pequenas mudanças e rode os testes após cada alteração. Essa abordagem minimiza o risco de quebrar o sistema de maneira significativa e facilita o diagnóstico de erros.

TESTES AUTOMATIZADOS

3. Mantenha Testes Atualizados: Quando alteramos o comportamento do sistema intencionalmente, é importante ajustar os testes para refletir as mudanças. Testes desatualizados podem gerar falsos positivos e dificultar o processo de refatoração.

4. Automatize ao Máximo: Executar testes manualmente é demorado e propenso a erros. Configure seu ambiente para rodar testes automaticamente após cada alteração no código, o que torna a refatoração mais rápida e segura.

Em resumo, a refatoração se torna mais segura, rápida e eficaz com **testes automatizados** robustos. Esses testes não só garantem que o **comportamento original seja preservado,** mas também trazem confiança e liberdade para que o código evolua de maneira contínua, sem comprometer a funcionalidade existente.

Obrigado!

Alguma Pergunta ?

willian_brito00@hotmail.com
linkedin.com/in/willian-ferreira-brito
github.com/willian-brito

REFATORAÇÃO