Capítulo 9 – Refactoring

1. Introdução

Refactoring refere-se a modificações no código que melhoram sua legibilidade, organização e manutenibilidade, sem alterar o funcionamento do sistema.

A necessidade de refatoração surge devido à evolução natural do software. Segundo as Leis de Lehman, sistemas de software envelhecem com o tempo, tornando-se mais complexos e difíceis de manter. O refactoring combate esse envelhecimento ao melhorar a estrutura do código sem modificar sua funcionalidade.

O conceito foi formalizado por Martin Fowler em 2000, com um catálogo de técnicas que ajudam a melhorar código existente. Ele enfatiza que refactoring deve ser feito continuamente, para evitar acúmulo de "dívida técnica".

2. Principais Técnicas de Refactoring

2.1 Extração de Método

Separa trechos de código repetitivos ou longos em métodos menores e mais legíveis.

Exemplo no mercado: Em um sistema de folha de pagamento, métodos longos de cálculo de salário podem ser divididos em métodos menores, melhorando a clareza e reutilização do código.

Exemplo Antes:

```
void calcularSalario() {
  double imposto = salario * 0.2;
  double bonus = salario * 0.1;
  salario = salario - imposto + bonus;
}
```

Após Refatoração:

```
void calcularSalario() {
  double imposto = calcularImposto();
  double bonus = calcularBonus();
  salario = salario - imposto + bonus;
}

double calcularImposto() { return salario * 0.2; }
double calcularBonus() { return salario * 0.1; }
```

2.2 Inline de Método

Remove métodos desnecessários e insere seu código diretamente no chamador.

Exemplo no mercado: Em uma startup que deseja otimizar seu código, pode ser necessário remover métodos supérfluos para melhorar a performance.

Exemplo Antes:

```
double calcularImposto() { return salario * 0.2; }
void calcularSalario() { salario = salario - calcularImposto(); }
```

Após Refatoração:

```
void calcularSalario() { salario = salario - (salario * 0.2); }
```

2.3 Movimentação de Método

Transfere um método para outra classe quando ele acessa mais dados de outra classe do que da própria.

Exemplo no mercado: Em um sistema ERP, pode ser necessário mover métodos de cálculo fiscal para uma classe específica de tributação.

Exemplo Antes:

```
class Cliente {
  Endereco endereco;
  String getCep() { return endereco.cep; }
}
```

Após Refatoração:

```
class Endereco {
  String cep;
  String getCep() { return cep; }
}
class Cliente {
  Endereco endereco;
}
```

2.4 Extração de Classe

Quando uma classe tem muitas responsabilidades, uma parte dela pode ser movida para uma nova classe.

Exemplo no mercado: Em um sistema de atendimento ao cliente, separar informações de contato da classe Cliente pode facilitar a organização dos dados.

Exemplo Antes:

```
class Cliente {
  String nome;
  String telefone;
  String email;
}
```

Após Refatoração:

```
class Contato {
  String telefone;
  String email;
}

class Cliente {
  String nome;
  Contato contato;
}
```

2.5 Renomeação de Método ou Variável

Melhora a clareza do código alterando nomes pouco intuitivos.

Exemplo no mercado: Em um sistema financeiro, renomear métodos de cálculo de taxas pode evitar erros e facilitar auditorias.

Exemplo Antes:

```
void c() { /* código */ }
```

Após Refatoração:

```
void calcularTotal() { /* código */ }
```

3. Code Smells (Indícios de Código Ruim)

Code smells são sinais de que um código precisa ser refatorado. Alguns exemplos incluem:

- **Código Duplicado:** Pode ser eliminado com **Extração de Método**.
- **Métodos Longos:** Melhorados com **Extração de Método** para dividir em partes menores.
- **Classes Grandes:** Podem ser divididas usando **Extração de Classe**.
- **Feature Envy:** O método pode ser movido para outra classe com **Movimentação de Método**.
- **Variáveis Globais:** Devem ser encapsuladas em classes específicas.

4. Estratégias para Aplicação de Refactoring

4.1 Refactoring Oportunista

- Realizado durante a implementação de novas funcionalidades.
- Pequenos ajustes no código para evitar o acúmulo de problemas.

4.2 Refactoring Planejado

- Realizado como uma **tarefa separada**, em situações onde grandes melhorias na estrutura do sistema são necessárias.

Recomenda-se combinar ambos os tipos para manter o código saudável!

5. Divida Técnica

Dívida técnica é um termo cunhado por Ward Cunningham, em 1992, para designar os problemas técnicos que podem dificultar a manutenção e evolução de um sistema. A ideia da dívida técnica é que, se esses problemas não forem resolvidos, futuramente exigirão um custo maior para correção, representado pelos "juros" da dívida. Isso se manifesta na dificuldade de manutenção, maior tempo para corrigir bugs e riscos ao implementar novas funcionalidades.

Por exemplo, se um módulo do sistema contém dívida técnica, adicionar uma nova funcionalidade pode levar 3 dias, enquanto sem a dívida levaria apenas 2 dias. Esse dia extra representa o "juros" da dívida. Remover a dívida pode demandar mais tempo inicialmente, mas, a longo prazo, facilita o desenvolvimento e reduz custos futuros.