Introdução a refatoração

Refatoração e Projeto de Software

A **refatoração** é o processo de melhorar a estrutura interna do código sem alterar seu comportamento externo. Essa prática é essencial para manter o código limpo, legível e fácil de manter, promovendo um design de software sustentável.

Benefícios da Refatoração

- **Redução da Complexidade Técnica**: Torna o código mais simples e direto.
- **Minimização da Dívida Técnica**: Facilita a manutenção e evolução do software no longo prazo.
- **Melhoria da Legibilidade**: Aumenta a clareza e facilita a colaboração entre desenvolvedores.

Boas Práticas para Refatoração

1. **Integre ao Ciclo de Desenvolvimento**
   Realize refatorações constantemente durante o desenvolvimento para evitar acúmulo de problemas.
2. **Trabalhe em Pequenos Passos**
   Faça mudanças incrementais e seguras para evitar a introdução de erros.
3. **Utilize Ferramentas de Teste**
   Certifique-se de que o comportamento do software permanece consistente usando testes automatizados.
4. **Priorize o Código Crítico**
   Concentre-se em áreas do código que são mais usadas ou mais propensas a erros.

A refatoração, quando bem aplicada, é uma ferramenta poderosa para melhorar a qualidade do software e garantir sua sustentabilidade ao longo do tempo.

Princípios de Refatoração

Por que refatorar?

A refatoração é essencial para garantir a qualidade e sustentabilidade de um projeto de software. Sem ela, o código tende a degradar ao longo do tempo, tornando-se difícil de manter e evoluir. Adotar práticas de refatoração preserva a consistência e o formato do projeto.

- **Melhora a compreensão do código**: Refatorar valida o entendimento sobre o sistema e esclarece detalhes antes obscuros, tornando o código mais acessível a toda a equipe.
- **Ajuda a encontrar bugs**: A refatoração aumenta o entendimento do código, tornando falhas de design e outros problemas mais evidentes e fáceis de corrigir.
- **Cria um ciclo virtuoso**: Boas práticas de refatoração geram melhorias contínuas, tornando o código mais organizado e facilitando a implementação de novos recursos e correções, criando um ambiente de desenvolvimento mais eficiente.
- **Acelera o desenvolvimento**: Refatoração melhora a estrutura, legibilidade e reduz falhas, resultando em maior qualidade de software e um ambiente propício para progresso constante.

Quando refatorar?

A refatoração pode ser realizada em momentos específicos ou de forma contínua, sendo mais eficaz quando aplicada de forma gradual durante o desenvolvimento. Ela traz melhores resultados quando feita de maneira incremental e consistente.

Regra dos três:
A "Regra dos Três" ajuda a determinar quando refatorar o código:

1ª vez: Implemente sem refatorar.
2ª vez: Duplique o código, mesmo sabendo que isso deve ser evitado.
3ª vez: Refatore.

Essa abordagem evita refatorações prematuras, permitindo primeiro entender o problema, depois implementar e, por fim, refatorar para melhorar a estrutura do código. O objetivo é realizar a refatoração apenas quando necessário, após a completa compreensão do problema.

Situações ideais para refatorar:

- **Dificuldade para modificar o código**: Se a alteração for difícil de implementar, refatore primeiro. Isso facilita a modificação e a torna mais eficiente.
- **Adição de novas funcionalidades**: Refatore antes de codificar novas funcionalidades. Isso melhora o entendimento do projeto e prepara a estrutura para a integração da nova funcionalidade.
- **Correção de bugs**: Bugs frequentemente indicam falhas de design. Refatorar o código ajuda a eliminar essas falhas e torna o sistema mais robusto.
- **Revisões de código**: As revisões de código são ótimas oportunidades para refatorar. Elas promovem o compartilhamento de conhecimento e permitem ajustes imediatos no código, tornando o projeto mais coeso e bem estruturado.

Problemas com refatoração

Embora a refatoração seja uma prática poderosa, ela não é uma solução mágica e não é aplicável em todas as situações. Em alguns casos, pode até ser prejudicial ao projeto. Além disso, as limitações da refatoração ainda não são totalmente compreendidas. A seguir, estão alguns problemas e limitações associadas à refatoração:

- **Banco de dados**: Aplicações frequentemente são fortemente acopladas aos esquemas de banco de dados. Alterações no modelo de dados podem exigir modificações no código e vice-versa. Para mitigar esse risco, é altamente recomendável usar uma camada de indireção entre o banco de dados e a aplicação, facilitando mudanças sem comprometer o sistema.
- **Mudanças em interfaces**: Refatorar interfaces pode ser problemático, especialmente se a interface for pública. Uma das vantagens dos objetos é poder alterar a implementação sem modificar a interface. No entanto, algumas refatorações impactam diretamente as interfaces, o que pode ser problemático se você não tem controle sobre os elementos que a utilizam.
  - **Como lidar com mudanças em interfaces publicadas**: Uma abordagem segura é manter a interface antiga junto com a nova até que todos os usuários migrem para a nova versão. Além disso, a implementação do método antigo pode ser ajustada para chamar o novo método.
  - **Princípio**: Evite publicar interfaces prematuramente. Só faça isso quando for realmente necessário.
- **Quando não refatorar**:
  - **Código extremamente complexo e instável**: Em alguns casos, o código está tão confuso e instável que, apesar de ser possível refatorá-lo, é mais vantajoso recomeçar do zero.
  - **Deadline próximo**: Refatoração pode melhorar a produtividade a longo prazo, mas leva tempo até que seus benefícios se tornem visíveis. Se o prazo do projeto estiver muito próximo, refatorar pode atrasar a entrega, tornando o prazo mais difícil de cumprir. Nesses casos, uma refatoração tardia, quando o projeto estiver mais estável, pode ser mais vantajosa.

Refatoração e projeto

A refatoração complementa o projeto de software, sendo uma prática contínua que ocorre ao longo do tempo. Tradicionalmente, o projeto é visto como uma atividade que precede a codificação. Com a refatoração, o projeto se transforma em uma consequência das operações realizadas sobre um sistema em funcionamento. A combinação de projetar para dividir responsabilidades e refatorar para aprimorar o projeto é a abordagem mais eficaz.

Mentalidade e abordagem durante o desenvolvimento:

- **Sem refatoração**:
  - Existe uma pressão por um projeto "perfeito" desde o início.
  - Desenvolvedores temem que mudanças no projeto serão caras e difíceis, o que gera resistência a modificações.
- **Com refatoração**:
  - O foco é encontrar uma solução plausível, não perfeita.
  - À medida que a solução se desenvolve, novas ideias surgem e o projeto evolui de forma mais flexível.

Consequências a longo prazo no design e manutenção:

- **Projetos sem refatoração**:
  - São mais rígidos e difíceis de adaptar a mudanças, exigindo planejamento meticuloso para futuras evoluções. Isso os torna caros e difíceis de manter.
- **Projetos com refatoração**:
  - São mais simples e flexíveis, permitindo adaptação fácil a novas necessidades e facilitando a manutenção e evolução contínuas.

Refatoração e performance

Refatorações tornam o código mais fácil de entender, mas podem reduzir o desempenho temporariamente. Contudo, um código bem estruturado facilita futuras melhorias de performance (tuning).

Soluções para desempenho de software:

1. **Decomposição em componentes**:
   Divida o sistema em componentes com limites definidos para tempo e consumo. Isso pode melhorar a organização, mas mal planejado pode afetar o desempenho.
2. **Atenção ao desempenho em mudanças**:
   Melhorias constantes de desempenho podem prejudicar a clareza do código. Melhorias gerais no código são ineficazes; foque nos pontos críticos.
3. **Fatoração antes de otimização**:
   Construa o software de forma bem estruturada primeiro, otimize depois. Use ferramentas para identificar hot-spots e realize otimizações em pequenos passos.

   Vantagens:
   - Mais tempo para otimizar.
   - Melhor precisão nas otimizações.

Maus-cheiros de código

"Maus-cheiros" de código são pontos onde os princípios de bom design não são seguidos ou podem ser melhorados. Identificar esses pontos é essencial para a refatoração e a melhoria contínua do projeto. O julgamento do refatorador é fundamental para decidir as operações que devem ser aplicadas.

1. Código Duplicado
Códigos repetidos em diferentes partes do projeto podem ser unificados, o que melhora a manutenção.

- **Operações aplicáveis**:
  - Extrair método (para trechos duplicados em uma classe).
  - Extrair método e puxar para cima (quando duplicado em classes irmãs).
  - Substituir algoritmo por um mais claro.

2. Método Longo
Métodos longos são difíceis de entender e manter. É melhor dividi-los em métodos menores, com nomes claros.

- **Operações aplicáveis**:
  - Extrair método (para reduzir o tamanho).
  - Introduzir objeto-parâmetro (para reduzir parâmetros longos).
  - Decompor expressões condicionais e loops.

3. Classe Grande
Classes com muitas variáveis indicam baixa coesão, pois fazem mais do que deveriam.

- **Operações aplicáveis**:
  - Extrair classe (para agrupar variáveis relacionadas).
  - Extrair subclasse (para variáveis que fazem sentido em uma subclasse).
  - Extrair interface (para isolar o uso das classes pelos clientes).

4. Longa Lista de Parâmetros
Listas de parâmetros longas tornam o código difícil de entender e manter.

- **Operações aplicáveis**:
  - Substituir parâmetro por método (acessando dados através de objetos).

5. Mudanças Divergentes
Mudanças em diferentes aspectos de uma classe (como regras de negócios e persistência) indicam que ela está mal estruturada.

- **Operações aplicáveis**:

- Extrair classe (para separar diferentes tipos de mudanças em classes distintas).

6. Cirurgia com Rifle
Quando uma mudança afeta várias classes, pode-se esquecer de algum detalhe. Esse problema, conhecido como "cirurgia com rifle", dificulta o controle sobre as alterações.

- **Operações aplicáveis**:
  - Mover método ou mover campo (para centralizar variações em uma única classe).
  - Incorporar classe (para agrupar comportamentos em uma estrutura coesa após mover métodos e campos).

7. Inveja de Recursos
Quando métodos de uma classe acessam atributos de outra classe de maneira excessiva, isso é um sinal de "inveja de recursos". A classe depende demais de outra para acessar dados.

- **Operações aplicáveis**:
  - Mover método (trazer o método para a classe correta).
  - Extrair método / mover método (quando a inveja ocorre apenas em um trecho específico do código).

8. Aglomerados de Dados
É comum que conjuntos de dados apareçam juntos em várias partes do código. Transformá-los em objetos pode melhorar a estrutura do código.

- **Operações aplicáveis**:
  - Extrair classe (para agrupar os dados em uma única estrutura).
  - Introduzir objeto-parâmetro (para transformar parâmetros de um método em um objeto, simplificando a assinatura).

9. Obsessão Primitiva
O uso excessivo de tipos de dados primitivos, como `int`, `float`, `boolean`, pode resultar em uma estrutura de código frágil. Transformar esses tipos em objetos ou classes pode melhorar a legibilidade e a manutenção.

- **Operações aplicáveis**:
  - Trocar dado por objeto (para substituir dados primitivos por objetos).
  - Trocar tipo código por classe (quando os dados representam um código).
  - Trocar tipo código por subclasses ou usar State/Strategy (quando os dados afetam comandos condicionais).
  - Extrair classe (para agrupar dados relacionados).
  - Introduzir parâmetro objeto (para agrupar dados em um único parâmetro de método).

10. Instruções `switch`
Instruções `switch` repetidas em várias partes do código indicam uma oportunidade para usar polimorfismo, o que torna o código mais flexível e reduz a duplicação.

- **Operações aplicáveis**:
  - Extrair método / Mover método (para mover o comando `switch` para um local adequado).
  - Trocar tipo por subclasse ou usar State/Strategy (para substituir a lógica `switch` por polimorfismo).

11. Hierarquias de Herança Paralelas
Esse problema ocorre quando, ao adicionar uma subclasse em um ramo da hierarquia de herança, é necessário replicá-la em outro ramo. Isso indica duplicação de código e dificulta a manutenção da estrutura do sistema.

- **Operações aplicáveis**:
  - Mover método e mover campo (para eliminar dependências entre as classes, centralizando a lógica necessária em uma única classe).

12. Classe Preguiçosa
Semelhante à "cirurgia com rifle", as classes preguiçosas geram duplicação de código, especialmente quando instâncias de uma hierarquia referenciam instâncias de outra. Isso leva à criação de subclasses redundantes em diferentes ramos.

- **Operações aplicáveis**:
  - Mover método e mover campo (para remover dependências entre as classes e evitar duplicação).

13. Generalidade Especulativa
Este "mau cheiro" ocorre quando o projetista cria classes ou métodos excessivamente genéricos, antecipando funcionalidades que podem nunca ser necessárias. Isso torna o código difícil de entender e manter.

- **Operações aplicáveis**:
  - Diminuir hierarquia (remover classes abstratas que não agregam valor).
  - Incorporar classe (eliminar delegações desnecessárias que tornam o código mais complexo).
  - Remover parâmetro (quando um parâmetro não é utilizado dentro de um método).
  - Quando um método ou classe é usado apenas em testes, pode ser um sinal de generalidade especulativa e de que o código pode ser simplificado.

14. Campo Temporário
Esse problema ocorre quando variáveis de instância de um objeto são utilizadas apenas em contextos específicos, gerando uma lista de parâmetros excessiva ou o uso de variáveis temporárias.

- **Operações aplicáveis**:
  - Extrair classe (para agrupar variáveis temporárias e fornecer um local centralizado para armazená-las).

15. Cadeias de Mensagens
Ocorre quando um objeto chama outro, que chama outro, criando uma sequência de chamadas de métodos (geralmente usando métodos Get). Isso é indicativo de alto acoplamento, onde a estrutura do projeto depende de chamadas encadeadas.

- **Operações aplicáveis**:
  - Ocultar delegação (para evitar a indireção indesejada).
  - Extrair método / mover método (para extrair o código repetido e movê-lo para um local mais adequado, rompendo a cadeia de chamadas).

16. Homem do Meio
Este problema acontece quando métodos de uma classe apenas delegam chamadas para métodos de outras classes, criando uma camada de indireção.

- **Operações aplicáveis**:
  - Remover homem do meio (eliminar a classe intermediária e chamar o método diretamente).
  - Introduzir método (quando o método de delegação faz muito pouco, traga seu comportamento para a classe que o chama).
  - Trocar delegação com herança (se houver um comportamento adicional, transforme a classe intermediária em uma subclasse do objeto real).

17. Intimidade Inapropriada
Quando uma classe tem acesso excessivo a partes privadas de outras classes, violando o princípio de encapsulamento e tornando o código excessivamente acoplado.

- **Operações aplicáveis**:
  - Mover método e mover campo (para reduzir a intimidade entre classes).
  - Mudar associação bidirecional para unidirecional (reduz o acoplamento entre as classes).
  - Extrair classe (para centralizar comportamentos comuns em uma nova classe).

- Ocultar delegação (para impedir que o cliente de uma classe conheça os detalhes de como a delegação é realizada).

18. Classes Alternativas com Interfaces Diferentes
Quando métodos com a mesma funcionalidade possuem assinaturas diferentes em classes diferentes, isso cria inconsistência no design e aumenta a complexidade.

- **Operações aplicáveis**:
  - Renomear método (para uniformizar as assinaturas dos métodos divergentes).
  - Mover método (para centralizar o comportamento nas classes que mais se beneficiariam dele).

19. Biblioteca de Classes Incompleta
Quando uma biblioteca de classes não fornece os recursos necessários, e o código não tem acesso para modificá-la.

- **Operações aplicáveis**:
  - Introduzir método estrangeiro (para adicionar comportamentos ausentes).
  - Introduzir extensão local (para criar soluções específicas dentro do projeto sem modificar a biblioteca original).

20. Classe de Dados
Essas classes contêm apenas dados e métodos de acesso, como métodos set e get. Elas são frequentemente usadas em excesso, o que pode causar problemas de encapsulamento.

- **Operações aplicáveis**:
  - Encapsular campo / coleções (para ocultar detalhes de implementação).
  - Remover método set (para evitar que campos sejam alterados diretamente de fora da classe).
  - Mover / extrair e ocultar método (para lidar com métodos set e get amplamente utilizados).

21. Herança Negada
Quando subclasses herdam métodos ou campos de suas classes-pai, mas não gostariam de tê-los, isso sugere que a hierarquia de herança está incorreta.

- **Operações aplicáveis**:
  - Descer método / campo (mover métodos e campos não utilizados para uma classe irmã).
  - Substituir herança por delegação (quando a subclasse não quer herdar a implementação de um método da superclasse, mas precisa usar a delegação para implementar sua própria lógica).

Exercício de fixação

Enunciado

Melhorar o código do exercício de fixação do documento Desenvolvimento Orientado a Testes (TDD) > Exercício de Fixação.

Código melhorado

```
import java.util.Stack;

public class ValidadorDeParenteses {
    public boolean ehBalanceada(String s) {
        Stack<Character> pilha = new Stack<>();

        // Mapeamento dos caracteres de fechamento para os de abertura
        java.util.Map<Character, Character> mapaDePareamentos = new
java.util.HashMap<>();
        mapaDePareamentos.put(')', '(');
        mapaDePareamentos.put(']', '[');
        mapaDePareamentos.put('}', '{');

        for (char c : s.toCharArray()) {
            // Se for um caractere de abertura, empurramos para a pilha
            if (mapaDePareamentos.containsValue(c)) {
                pilha.push(c);
            }
            // Se for um caractere de fechamento
            else if (mapaDePareamentos.containsKey(c)) {
                // Verifica se a pilha está vazia ou o topo da pilha não
corresponde
                if (pilha.isEmpty() || pilha.pop() !=
mapaDePareamentos.get(c)) {
                    return false;
                }
            }
        }

        // A string está balanceada se a pilha estiver vazia
        return pilha.isEmpty();
    }
}
```

Explicação das mudanças

1. Uso de mapa para pareamentos:

- Foi criado um `HashMap` que mapeia os caracteres de fechamento para sues respectivos caracteres de abertura (`')' -> '('`, `']' -> '['`, `'}' -> '{'`). Isso elimina a necessidade de múltiplos `if` no código e torna a verificação mais clara e eficiente.

2. Eliminação de código duplicado:

- Antes, o código verificava explicitamente cada tipo de delimitador dentro da lógica de controle. Com o mapa de pareamentos, a lógica fica centralizada em um único local.

3. Condição Simplificada para fechamento:

- Quando um caractere de fechamento é encontrado, ele é comparado diretamente com o topo da pilha usando o mapa, o que elimina a necessidade de uma série de `if` adicionais.

4. Legibilidade melhorada:

- A separação clara entre a verificação de abertura e fechamento torna o código mais intuitivo.

Com essas mudanças, o código fica mais modular, com menor duplicação, e mais fácil de entender e manter.