ESTUDO DE CASO – SOLID, PRINCÍPIO DA RESPONSABILIDADE ÚNICA E REFATORAÇÃO

A partir do código disponibilizado, por favor siga as orientações dadas no questionário e responda as questões apresentadas. O texto apresentado abaixo serve de apoio para os que não tem experiência com os princípios S.O.L.I.D e refatoração de código fonte, fornecendo uma introdução sobre o tema e um passo-a-passo para as refatorações de código fonte.

PRINCÍPIOS S.O.L.I.D E REFATORAÇÃO DE CÓDIGO FONTE

INTRODUÇÃO AOS PRINCÍPIOS S.O.L.I.D

Os Princípios SOLID são diretrizes de design de software que promovem a criação de sistemas robustos, flexíveis e de fácil manutenção. Esses princípios desempenham um papel fundamental na construção de software de qualidade. Ao seguir esses princípios, os desenvolvedores podem criar sistemas que são fáceis de entender, modificar e estender. Os benefícios incluem maior reutilização de código, menor acoplamento entre módulos, maior testabilidade e manutenibilidade, e uma arquitetura mais flexível e escalável. Tais princípios foram introduzidos por Robert C. Martin e se tornaram referências importantes na Engenharia de Software.

Princípio da Responsabilidade Única (Single Responsibility Principle - SRP): Este princípio estabelece que uma classe deve ter apenas uma única responsabilidade. Isso significa que uma classe deve ter um único motivo para mudar. Ao seguir o SRP, as classes se tornam mais coesas e seu código é mais fácil de entender, modificar e testar.

Princípio do Aberto/Fechado (Open/Closed Principle - OCP): O OCP enfatiza que as entidades de software (classes, módulos, etc.) devem ser abertas para extensão, mas fechadas para modificação. Isso significa que o comportamento de uma entidade pode ser estendido sem alterar seu código-fonte existente. O OCP promove um design flexível e evitando que modificações causem efeitos colaterais em outras partes do sistema.

Princípio da Substituição de Liskov (Liskov Substitution Principle - LSP): O LSP estabelece que as subclasses devem ser substituíveis por suas classes base, sem quebrar a integridade do sistema. Em outras palavras, uma classe derivada deve ser capaz de substituir perfeitamente sua classe base, sem causar comportamentos inesperados ou violações de contrato.

Princípio da Segregação de Interfaces (Interface Segregation Principle - ISP): O ISP afirma que as interfaces devem ser coesas e específicas para os clientes que as utilizam. Em vez de ter interfaces genéricas que abrangem várias funcionalidades, o ISP preconiza a criação de interfaces menores e mais especializadas, adaptadas às necessidades dos clientes individuais. Isso promove a modularidade e a reutilização de código.

Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP): O DIP propõe que as dependências entre módulos devem ser baseadas em abstrações e não em implementações concretas. Ele incentiva a inversão do controle, permitindo que módulos de alto nível dependam de abstrações em vez de módulos de baixo nível. Isso facilita a substituição de implementações e torna o sistema mais flexível e testável.

Dentre esses princípios, o princípio da responsabilidade única se destaca por ser de simples adoção e ter um importante impacto na diminuição da complexidade do código fonte. Assim, Martin (2008) identificou os seguintes sinais de que um método ou classe está violando o princípio da responsabilidade única:

- O método ou classe faz coisas que não se relacionam entre si. Por exemplo, um método que carrega dados de um banco de dados e também calcula um valor não está cumprindo o princípio da responsabilidade única;
- O método ou classe é difícil de entender. Se um método ou classe tem muitas responsabilidades, pode ser difícil entender o que ele faz;
- O método ou classe é difícil de testar. Se um método ou classe tem muitas responsabilidades, pode ser difícil escrever testes para ele;
- O método ou classe é difícil de manter. Se um método ou classe tem muitas responsabilidades, pode ser difícil fazer alterações nele sem quebrar outras partes do código.

Nesse sentido, uma classe que representa um usuário e também gerencia a autenticação do usuário não está cumprindo o princípio da responsabilidade única. A classe deve ser dividida em duas classes, uma para representar o usuário e outra para gerenciar a autenticação. Uma classe que representa um produto e também gerencia as vendas do produto não está cumprindo o princípio da responsabilidade única. A classe deve ser dividida em duas classes, uma para representar o produto e outra para gerenciar as vendas.

INTRODUÇÃO À REFATORAÇÃO DE CÓDIGO FONTE PARA CONFORMIDADE COM O PRINCÍPIO DA RESPONSABILIDADE ÚNICA

É importante notar que o seguimento do princípio da responsabilidade única está intrinsecamente ligado com a prática de refatoração de código fonte. Refatorar código é alterar sua estrutura, de forma metódica e controlada, sem alterar seu comportamento. A refatoração desempenha um papel crucial no desenvolvimento de software, contribuindo para a qualidade, manutenibilidade e evolução contínua do sistema. Ao melhorar a estrutura interna do código, a refatoração promove um design mais limpo, reduz a complexidade e facilita a identificação e correção de problemas. É uma técnica indispensável para manter o software saudável e adaptável ao longo do tempo.

Nesse âmbito, um código que funciona é mantido funcionando, mas sua estrutura e alterada para que seu entendimento e manutenção se tornem mais fáceis. De acordo com o livro Refatoração de Martin Fowler (2002), as refatorações mais utilizadas para resolver o problema da quebra do princípio da responsabilidade única são:

- Extração de método: A extração de método é uma técnica de refatoração que envolve a criação de um novo método a partir de um trecho de código existente em um método mais longo. Isso ajuda a melhorar a legibilidade, modularidade e reusabilidade do código.
- Método de Delegação: A técnica de refatoração conhecida como "Método de Delegação" (Delegate Method) é usada quando você quer delegar a responsabilidade de uma operação a outro objeto.
- Princípio da Segregação de Interfaces: O Princípio da Segregação de Interfaces (ISP) é um dos cinco princípios SOLID e sugere que uma classe não deve ser forçada a implementar interfaces que ela não utiliza. Isso significa que as interfaces devem ser específicas e conter apenas métodos que são relevantes para as classes que as implementam.

Apresenta-se aqui um passo-a-passo de cada uma das refatorações necessárias para e suas finalidades.

Para a extração de método, o passo-a-passo apresentado foi:

1. Escolha do Trecho:

    a. Identifique um trecho de código em um método que realize uma operação específica ou represente uma responsabilidade distinta.
2. Crie um Novo Método:
    a. Escolha um nome descritivo para o novo método que represente claramente a operação ou responsabilidade que será encapsulada.
    b. Declare o novo método no escopo da classe.
3. Parâmetros do Novo Método:
    a. Se necessário, determine os parâmetros necessários para o novo método. Esses parâmetros são geralmente as variáveis locais do método original que o novo método precisará para executar sua tarefa.
4. Mova o Trecho de Código:
    a. Copie o trecho de código identificado para o novo método. Certifique-se de incluir todas as variáveis necessárias como parâmetros.
5. Refatoração do Método Original:
    a. Substitua o trecho de código no método original com uma chamada para o novo método recém-criado.
6. Ajustes e Testes:
    a. Verifique se o código ainda funciona corretamente após a extração do método.
    b. Faça ajustes se necessário, e certifique-se de que o método original e o novo método estão funcionando conforme o esperado.

Já para o método de delegação:

1. Identificação da Operação a Delegar
    a. Identifique a operação ou responsabilidade específica dentro de um método que você deseja delegar a outro objeto.
2. Escolha ou Criação do Objeto Delegado
    a. Selecione um objeto existente ou crie um novo objeto que será responsável por realizar a operação delegada.
3. Criação do Novo Método Delegado
    a. No objeto delegador, crie um novo método que realize a operação desejada. Esse será o método delegado.
4. Parâmetros do Novo Método Delegado:

    a. Se necessário, determine os parâmetros necessários para o novo método delegado. Esses parâmetros podem incluir variáveis locais do método original que são necessárias para realizar a operação.
5. Chamada ao Novo Método Delegado:
    a. Substitua a lógica da operação no método original com uma chamada ao novo método delegado no objeto delegado.
6. Ajustes e Testes:
    a. Certifique-se de que o código ainda funciona corretamente após a implementação do método de delegação.
    b. Faça ajustes se necessário, e teste para garantir que a delegação está ocorrendo conforme o esperado.

Por fim, para o princípio da segregação de interfaces (ISP), o passo-a-passo apresentado é:

1. Identificação de Interfaces Monolíticas:
    a. Identifique interfaces em seu sistema que estão se tornando grandes e monolíticas, contendo métodos que podem não ser necessários para todas as classes que a implementam.
2. Divisão em Interfaces Menores:
    a. Divida a interface monolítica em interfaces menores e mais específicas, cada uma contendo métodos relacionados entre si.
3. Implementação Seletiva:
    a. Faça com que as classes implementem apenas as interfaces que contêm métodos relevantes para elas. Se uma classe não precisa de um determinado método, ela não deve ser forçada a implementar a interface que o contém.
4. Refatoração de Código:
    a. Altere as classes para implementar apenas as interfaces relevantes para suas necessidades. Remova as implementações desnecessárias.
5. Criar Interfaces de Propósito Único:
    a. Crie interfaces com propósitos únicos e específicos. Cada interface deve representar uma responsabilidade ou conjunto coeso de funcionalidades.
6. Garantir Cumprimento de Contratos:

a. Certifique-se de que, apesar da divisão de interfaces, todas as classes ainda cumpram os contratos esperados pelas interfaces que implementam.

7. Testes:
    a. Realize testes para garantir que a refatoração não introduziu quebras de funcionalidade. Certifique-se de que cada classe continua a funcionar conforme o esperado.

A RELEVÂNCIA DOS TESTES COMO APOIO AO PROCESSO DE REFATORAÇÃO DE CÓDIGO FONTE

É importante notar que as refatorações apresentadas demandam testes. Isso ocorre pela própria natureza do processo de refatoração: um código sem testes executáveis não pode ser facilmente verificado quanto ao atendimento dos requisitos funcionais e não-funcionais quando refatorado, a não ser por testes manuais. Isso mostra a importância de uma suíte de testes também como complemento ao processo de refatoração, uma vez que geram segurança para introdução de modificações no código fonte.

Os testes de software têm como objetivo verificar se o software atende aos requisitos especificados, identificar e corrigir erros e garantir que o sistema funcione corretamente em diferentes condições e cenários de uso. Eles desempenham um papel fundamental na melhoria da qualidade do software e no aumento da confiabilidade do sistema.

Conforme definido por Beizer (1995, p. 22), "o objetivo dos testes é encontrar o maior número possível de erros com o menor esforço possível". Os testes de software permitem que os desenvolvedores identifiquem problemas de funcionamento, verifiquem a conformidade com os requisitos do sistema e validem a corretude do software antes de ser lançado em produção.

Além disso, os testes de software também contribuem para a redução de custos a longo prazo. De acordo com Boehm (1981, p. 106), "o custo de encontrar e corrigir um erro após a liberação é de 100 a 1.000 vezes maior do que o custo de encontrá-lo durante a fase de design". Isso destaca a importância de investir em testes de software desde as fases iniciais do processo de desenvolvimento, a fim de evitar problemas onerosos e potencialmente prejudiciais posteriormente.

Existem várias abordagens de testes de software, cada uma com seus objetivos específicos e técnicas associadas. Destacam-se aqui os testes unitários, que são

realizados nas menores unidades de código, como funções ou métodos individuais, para verificar se eles produzem os resultados esperados, e o testes de integração ou funcionais, que verificam a interação entre diferentes componentes do sistema e garantem que eles funcionem corretamente em conjunto.