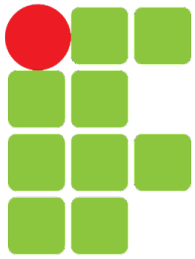

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PIAUÍ

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
ENGENHARIA DE SOFTWARE III
PROF.: Ely Miranda

ATIVIDADE 05

PESQUISE A RESPEITO DO INTERFACE SEGREGATION PRINCIPLE (ISP) E RESPONDA AS QUESTÕES ABAIXO:

1. Qual a principal imagem relacionada ao princípio e qual a explicação sobre ela?

O princípio da separação de interfaces diz que as interfaces que são muito grossas devem ser divididas em interfaces menores e mais específicas, para que os clientes de interfaces pequenas só saibam sobre os métodos que precisam para fazer seu trabalho. Como resultado, quando você altera um método de interface, os clientes que não usam esse método não devem mudar.

2. Por que devemos segregar implementações através do uso de interfaces?

A segregação de implementações através do uso de interfaces traz vários benefícios no contexto da engenharia de software. Alguns desses benefícios incluem: boas práticas de design, testabilidade melhorada, modularidade, manutenção simplificada, redução do acoplamento e facilita a substituição de implementações. Por exemplo, interfaces bem definidas simplificam a manutenção do código. Se uma classe precisa ser modificada ou se tornar obsoleta, outras classes que dependem apenas da interface não são afetadas, desde que a interface permaneça inalterada.

A segregação de implementações através de interfaces é uma prática valiosa que contribui para a construção de sistemas mais flexíveis, modulares e fáceis de manter. Isso ajuda a garantir que o software seja adaptável a mudanças e promove um design mais coeso e orientado a contratos.

Quando se trata do princípio de segregação de interfaces, seguir este princípio ajuda o sistema a permanecer flexível ao fazer alterações na lógica de trabalho e adequado para refatoração.

O princípio da separação de interfaces reduz a complexidade de manutenção e desenvolvimento da aplicação. Quanto mais simples e minimalista for a interface utilizada, menos recursos são necessários para implementá-la em novas classes e haverá, assim, menos razão para modificá-la. Mesmo no caso de surgir alguma modificação em algum ponto da aplicação, será muito mais fácil implementá-la.

3. Podemos dizer que esse princípio é correlato ao Single Responsibility Principle pelo fato de que classes e interfaces devem ter um único “foco” (SRP) e que as interfaces com métodos não correlatos devem ser segregadas (ISP)?

Sim, o Interface Segregation Principle (ISP) e o Single Responsibility Principle (SRP) são princípios de design de software que compartilham algumas semelhanças a nível de implementação.

O Single Responsibility Principle (SRP) afirma que uma classe deve ter apenas uma responsabilidade. Isso significa que a classe deve ter apenas uma razão para mudar. Por outro lado, o Interface Segregation Principle (ISP) afirma que uma classe não deve ser forçada a depender de métodos que ela não usa. Isso significa que as interfaces devem ser coesas e ter apenas os métodos necessários para a classe que as implementa.

Embora esses princípios sejam diferentes, eles estão relacionados. O ISP ajuda a implementar o SRP, aumentando a coesão e reduzindo o acoplamento. Ao seguir o ISP, pode-se criar interfaces menores e mais específicas, o que ajuda a manter o código organizado e fácil de entender.

4. Comente as seguintes frases à luz do ISP:

a. “Nenhum cliente deve ser forçado a depender de métodos que ele não usa”;

O ISP visa garantir que as interfaces de uma classe sejam específicas e coesas, para que os clientes importem apenas os métodos que eles utilizam. A frase “nenhum cliente deve ser forçado a depender de métodos que ele não usa” é uma das formas de expressar a afirmação de algo já demonstrado.

Em outras palavras, o ISP sugere que as interfaces devem ser divididas em partes menores e mais específicas, para que os clientes não sejam obrigados a implementar métodos que eles não precisam. Isso ajuda a evitar a dependência de funcionalidades desnecessárias e a manter a coesão da classe.

Se esse princípio for violado, um cliente que usa uma interface com todos os seus métodos depende de métodos que ele não usa e, portanto, é suscetível a mudanças nesses métodos. Como resultado, chegamos a uma relação estreita entre as diferentes partes da interface, que pode não estar relacionada com a sua implementação.

b. “Classes não devem ser forçadas a implementar interfaces que não usam”;

O ISP também afirma que uma classe não deve ser forçada a implementar interfaces que ela não usa. Em outras palavras, uma classe deve ter a flexibilidade de implementar apenas os métodos que são relevantes para ela, evitando assim a imposição de uma carga desnecessária.

A frase acima destaca a importância de se criar interfaces específicas e coesas, adaptadas às necessidades das classes que as implementam. Se uma classe é obrigada a implementar métodos que não têm significado ou utilidade para ela, isso viola o ISP.

Um exemplo prático seria uma interface que contenha diversos métodos, e uma classe que precisa implementar essa interface apenas para um subconjunto desses métodos. Isso pode levar a classes com implementações vazias ou a métodos que não fazem sentido para a classe em questão.

PESQUISE A RESPEITO DO DEPENDENCY INVERSION PRINCIPLE (DIP) E RESPONDA ÀS QUESTÕES ABAIXO:

5. Qual a principal imagem relacionada ao princípio e qual a explicação sobre ela?

A inversão de dependência é um princípio que sugere que módulos de alto nível não devem ser dependentes de módulos de baixo nível no processo de desenvolvimento de software, e que ambos devem ser dependentes de abstrações. Este princípio inclui as seguintes ideias principais: os módulos de alto nível não devem depender dos detalhes de implementação dos módulos de nível inferior, enquanto que os módulos de segundo nível devem ser dependentes de abstrações de alto nível; e as abstrações não devem depender de detalhes.

O princípio desta forma de programar aumenta características importantes do software desenvolvido, como flexibilidade, facilidade de manutenção e estabilidade. Ele também aumenta a reutilização do código e limita o impacto das alterações, criando menos dependências.

6. O que você entende por programar para Interfaces?

As interfaces permitem que você declare um conjunto de funções que devem ser implementadas em qualquer objeto que deseje usar essa interface. São úteis quando se deseja dividir seu código em componentes menores que podem ser usados separadamente uns dos outros. Eles permitem que se forneça um conjunto específico de funções que podem ser usadas por qualquer objeto que ofereça suporte a essa interface, independentemente de sua implementação específica.

As interfaces permitem que se forneça flexibilidade e escalabilidade ao seu código. Ao definir interfaces, pode-se especificar alguns conjuntos de funções e propriedades que devem ser implementadas em uma classe que usa essa interface.

Graças às interfaces, podemos garantir a intercambialidade de diferentes implementações de classe, o que é muito útil em situações em que queremos usar implementações diferentes da mesma funcionalidade. A implementação de uma interface pode ser modificada sem alterar o código que usa essa interface. Isso é especialmente útil ao projetar sistemas grandes e complexos.

Além disso, as interfaces permitem melhorar a legibilidade e a compreensão do código. Usando interfaces, fica claro quais métodos devem ser implementados na classe e com quais parâmetros e valores de retorno eles devem trabalhar.

Uma das principais razões para o uso de interfaces é melhorar a modularidade do código do programa. Essa modularidade do código permite melhor legibilidade, compreensão e reutilização. Um desenvolvedor pode trabalhar em componentes individuais de um programa de forma independente, sabendo que eles irão interagir corretamente uns com os outros de acordo com certas interfaces.

Em suma, trabalhar com interfaces permite abstrair de implementações específicas e criar componentes de software modulares e reutilizáveis. Eles permitem que você especifique o conjunto de recursos que devem ser fornecidos pela implementação, mas não especificam como esses recursos serão implementados. Isso simplifica o desenvolvimento e a manutenção de sistemas complexos porque permite substituir uma implementação de uma interface por outra sem alterar o código que usa essa interface.

8. Estude o conteúdo do link abaixo e explique como o DIP se aplicaria:

https://pt.stackoverflow.com/questions/101692/como-funciona-o-padr%C3%A3o-repository

No exemplo acima, é comentado a respeito de uma aplicação em PHP, através do framework Symphony, utilizando-se de um padrão de arquitetura conhecido como *repository pattern.*

O Repository Pattern é um padrão de projeto de software que atua como uma camada intermediária entre a lógica de negócios de um aplicativo e o armazenamento de dados. Ele é usado para abstrair a lógica de acesso a dados, permitindo que a lógica de negócios se concentre em operações de alto nível. O padrão é uma maneira de separar a lógica de acesso a dados da lógica de negócios, tornando o código mais modular e fácil de manter.

Por ser mais abstrato e fornecer uma interface mais genérica para acessar dados, o Repository Pattern é usado para encapsular a lógica de acesso a dados em uma classe separada, permitindo que a lógica de negócios se concentre em operações de alto nível. Em resumo, o padrão Repository é uma maneira de abstrair a lógica de acesso a dados em uma classe separada, permitindo que a lógica de negócios se concentre em operações de alto nível. Ele é usado para tornar o código mais modular e fácil de manter, e é frequentemente usado em conjunto com outros padrões de projeto para criar aplicativos mais modulares e fáceis de manter.

De modo geral, o padrão de repositório corresponde à separação entre lógica de negócios e aquisição de dados, independentemente da fonte de dados.

Neste contexto, a aplicação do princípio da inversão de dependências está mais do que certo. Ao invés de se utilizar um módulo que realiza a consulta no banco de dados diretamente, sendo uma clara evidência de dependência de informações que poderão ser mutáveis, o exemplo da classe Usuário cria uma camada entre uma consulta e o banco de dados, sem necessitar de um acesso direto.

O exemplo sugerido é usado para isolar o código de acesso a dados e manipulação de dados da lógica de negócios, proporcionando uma separação clara entre essas responsabilidades, reafirmando a definição do DIP; isso centraliza a lógica de acesso a dados em um único local, tornando mais fácil manter e modificar a forma como os dados são manipulados.

9. Você acha que esse princípio deveria ser um dos primeiros em um eventual "check-list" de princípios SOLID a aplicarmos?

Sim. O SOLID é um termo coletivo para cinco princípios de design de software para escrever código melhor e mais sustentável em linguagens de programação orientadas a objetos. O SOLID é apoiado por uma série de padrões de projeto como bons e bem testados garantes de que os princípios são cumpridos na prática.

Com base nisso, o DIP surge com a possibilidade de substituir facilmente módulos individuais por outros, alterando o módulo de dependência. Isso permite que se altere um módulo sem afetar os outros, levando a um código muito mais legível e abrindo portas para se empregar os demais princípios.

10. Numa visão geral, considera que se começarmos pelos 4 primeiros princípios SOLID, fatalmente o DIP já estaria implementado?

Sim; isso é possível levando em consideração que os quatro primeiros princípios SOLID podem ser vistos como práticas que naturalmente contribuem para a implementação do quinto princípio, o Dependency Inversion Principle (DIP).

Em conjunto, esses quatro princípios fornecem uma base sólida para a implementação bem-sucedida do DIP. Eles promovem um design de código que é mais extensível, menos propenso a alterações e mais adaptável a diferentes implementações.

O SRP ajuda a garantir que uma classe tenha uma única razão para mudar. Isso reduz a probabilidade de alterações em uma classe afetarem outras partes do sistema. Classes que têm uma única responsabilidade tendem a ser mais estáveis e menos propensas a mudanças. Isso facilita a inversão de dependências, pois mudanças em uma classe não afetarão várias partes do sistema.

O OCP incentiva o design de classes que são abertas para extensão, mas fechadas para modificação. Ao criar código que pode ser estendido sem modificar o código existente, você pode introduzir novas implementações sem impactar as classes existentes. Isso ajuda na inversão de dependências, pois as classes existentes não precisam ser alteradas quando novas implementações são introduzidas.

O LSP promove a substituição de objetos de uma classe derivada por objetos da classe base sem afetar a integridade do programa. Isso implica que as classes derivadas podem ser introduzidas sem que as classes clientes precisem ser alteradas. Essa flexibilidade contribui para a inversão de dependências, já que as classes dependentes não precisam ser modificadas quando novas implementações são introduzidas.

O ISP sugere que uma classe não deve ser forçada a implementar interfaces que não utiliza. Isso evita que uma classe dependa de funcionalidades que não são relevantes para ela. Isso contribui para a inversão de dependências, pois as classes podem depender apenas das interfaces que realmente precisam, reduzindo assim o acoplamento.