UNIVERSIDADE DO ESTADO DE MINAS GERAIS-UEMG
NÚCLEO ACADÊMICO DE TECNOLOGIA E ENGENHARIA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Engenharia de Software II

PRINCÍPIOS SOLID E PADRÕES DE PROJETO

Equipe
Anderson Veloso dos Santos
Guilherme Reis Ferreira
Junior César da Silva
Maria Andressa de Paula Silva
Rafael de Oliveira Romano
Ricardo Botelho Mariano

PASSOS/MG
2016

Princípios SOLID

SOLID é um acrônimo dos cinco primeiros princípios da programação orientada a objetos. Devem ser aplicados para se obter os benefícios da orientação a objetos, tais como ser fácil de se manter, adaptar e se ajustar às alterações de escopo; ser testável e de fácil entendimento; ser extensível para alterações com o menor esforço necessário; que forneça o máximo de reaproveitamento e que permaneça o máximo de tempo possível em utilização. São eles:

Letra	Sigla	Nome
S	SRP	Princípio da Responsabilidade Única
O	OCP	Princípio Aberto-Fechado
L	LSP	Princípio da Substituição de Liskov
I	ISP	Princípio da Segregação da Interface
D	DIP	Princípio da Inversão da Dependência

1- SRP (Single Responsibility Principle)

Princípio da Responsabilidade Única.

Este é o primeiro princípio, onde diz que uma classe deve ter apenas um motivo para ser modificada. Sendo assim, a classe deve ter uma única responsabilidade, já que a responsabilidade indica mudança. Uma classe com mais de um motivo para mudar possui mais de uma responsabilidade, e isso indica que ela não é coesa o que afronta os princípios da orientação a objeto.

Quando utilizado, podemos observar várias vantagens, tais como: coesão, facilidade de manutenção e compreensão, facilidade em reuso, alteração da responsabilidade não gera comprometimento em outras já que a classe possui uma única responsabilidade, não há acoplamento. O SRP é um dos mais importantes na orientação a objetos e com ele é possível projetar classes menores, mais coesas e de fácil entendimento.

2- OCP (Open Closed Principle)

Princípio do Aberto Fechado.

O OCP é mais um daqueles princípios de orientação a objetos que nos ajudam a eliminar design smells, possibilitando que nosso código ganhe em facilidade de manutenção e extensão.

DEFINIÇÃO:

“Entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação. ”

Quando eu precisar estender o comportamento de um código, eu crio código novo ao invés de alterar o código existente. E como isso é possível? Como adicionar comportamentos novos sem alterar o código existente? Isso mesmo: ABSTRAÇÃO!

Quando uma mudança causa uma série de mudanças em cascata, fica claro que nosso design não está bom pois, além de mais trabalho para alterarmos, ainda podemos nos esquecer de algumas partes do código.

O Princípio do Aberto/Fechado nos atenta para a aplicação de abstrações e polimorfismo, de forma consciente, garantindo que tenhamos um software mais flexível e, portanto, mais fácil de ser mantido.

3- LSP (Liskov Substitution Principle)

Princípio de Substituição de Liskov

O Princípio de Substituição de Liskov leva esse nome por ter sido criado por Barbara Liskov, em 1988. Sua definição mais usada diz que: “Classes derivadas devem poder ser substituídas por suas classes base”. “Se para cada objeto o1 do tipo S há um objeto o2 do tipo T de forma que, para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando o1 é substituído por o2 então S é um subtipo de T”.

Violar o LSP pode provocar comportamentos inesperados no software por suposições equivocadas quando ao funcionamento das classes derivadas de uma hierarquia de classes. Além disso, sua violação pode implicar em uma violação no OCP, causando todos os demais problemas consequentes desta.

Ao atender o Princípio de Substituição de Liskov (LSP), ou seja, ao garantir que as classes derivadas sejam completamente substituíveis por suas classes-base, todo código que utilizar a classe base será capaz de atender o OCP, facilitando a manutenção e extensão do software, além de ser um código mais seguro, livre de mau funcionamento como o mostrado no exemplo do quadrado. Resumindo: é importante prestar atenção na hierarquia de classes, fazer bom uso do polimorfismo e não esquecer que uma relação “É UM” se refere a COMPORTAMENTO.

4- ISP (Interface Segregation Principle)

Princípio da Segregação de Interface

O Princípio da Segregação de Interface trata da coesão de interfaces e diz que clientes não devem ser forçados a depender de métodos que não usam.

EXEMPLO DE VIOLAÇÃO:

Vejamos a seguinte interface:

```
1   public interface MembroDeTimeScrum
2   {
3       void PriorizarBacklog();
4       void BlindarTime();
5       void ImplementarFuncionalidades();
6   }
```

E agora temos as classes Dev, ScrumMaster e ProductOwner implementando a interface MembroDeTimeScrum:

```
1   public class Dev : MembroDeTimeScrum
2   {
3       public void PriorizarBacklog() { }
4       public void BlindarTime() { }
5
6       public void ImplementarFuncionalidades()
7       {
8           Console.Writeline("Codando e tomando café compulsivamente!!");
9       }
10  }
11
12  public class ScrumMaster : MembroDeTimeScrum
13  {
14      public void PriorizarBacklog() { }
15
16      public void BlindarTime()
17      {
18          Console.Writeline("Devs working! You shall not pass!!!!");
19      }
20
21      public void ImplementarFuncionalidades() { }
22  }
```

```
19
20      public class ProductOwner : MembroDeTimeScrum
21      {
22          public void PriorizarBacklog()
23          {
24              Console.Writeline("Priorizando backlog com base nas minhas nessecidades
25      de negócio");
26          }
27
28          public void BlindarTime() { }
29          public void ImplementarFuncionalidades() { }
30      }
31
```

Ao criarmos uma interface genérica demais, acabamos fazendo com que uma implementação, no caso Dev, não utilize certos métodos da interface. É o que acontece com os métodos PriorizarBacklog e BlindarTime, que não fazem nada, pois não são atribuições de um Dev e sim do ProductOwner e do ScrumMaster, respectivamente.

PROBLEMAS:

Suponhamos que alguma alteração seja necessária no método BlindarTime, que agora precisa receber alguns parâmetros. Dessa forma, somos obrigados a alterar todas implementações de MembroDeTimeScrum – Dev, ScrumMaster e ProductOwner – por causa de uma mudança que deveria afetar apenas a classe ScrumMaster.

Além disso, classes-cliente que dependiam de MembroDeTimeScrum terão que ser recompiladas e se estão em diversos componentes terão que ser redistribuídas. Algumas vezes desnecessariamente, pois nem sequer faziam uso do método BlindarTime!

Outro problema é que a implementação de métodos inúteis (chamados “degenerados”) pode levar à violação do LSP, pois alguém utilizando MembroDeTimeScrum poderia supor o seguinte:

```
1       foreach(var membro in membrosDeTimeScrum)
2         membro.ImplementarFuncionalidades();
```

No entanto, sabemos que apenas Dev executa o comportamento acima. Se a lista tivesse também objetos do tipo ScrumMaster ou ProductOwner, esses objetos não estariam realizando nada, ou pior, poderiam disparar alguma exceção, caso a implementação dos mesmos assim o fizesse.

RESOLVENDO A VIOLAÇÃO DO ISP:

A solução para o exemplo acima seria criamos interfaces mais específicas para que cada classe cliente dependa apenas do que realmente necessita. Por exemplo:

```
public interface FuncaoDeScrumMaster
{
    void BlindarTime();
}

public class ScrumMaster : FuncaoDeScrumMaster
{
    public void BlindarTime()
    {
        Console.Writeline("Devs working! You shall not pass!!!!");
    }
}
```

Com a alteração acima, a classe concreta ScrumMaster não precisa mais implementar métodos desnecessários e demais classes que dependiam de MembroDeTimeScrum apenas para utilizar BlindarTime podem agora depender da interface FuncaoDeScrumMaster.

A mesma ideia pode ser aplicada para as funções específicas de Dev e ProductOwner. Assim todos os clientes de MembroDeTimeScrum agora podem depender especificamente das interfaces que utilizam.

CONCLUSÃO:

O Princípio da Segregação de Interface nos alerta quanto à dependência em relação a “interfaces gordas”, forçando que classes concretas implementem métodos desnecessários e causando um acoplamento grande entre todos os clientes.

Ao usarmos interfaces mais específicas, quebramos esse acoplamento entre as classes clientes, além de deixarmos as implementações mais limpas e coesas.

5- DIP (Dependency Inversion Principle)

Princípio da Inversão de Dependência

Princípio da Inversão de Dependência procura manter o foco da tarefa de design no negócio, deixando este design independente ou desacoplado do componente que vai executar as tarefas de baixo nível que não fazem parte da modelagem do negócio.

Em respeito a este princípio, em vez de desenvolver o componente de baixo nível e a partir dele orientar o desenvolvimento do componente de alto nível, você define como vai ser a interação entre estes componentes (sempre privilegiando as necessidades de design do componente de alto nível), e daí o componente de baixo nível tem que ser desenvolvido respeitando esta definição de interação com o componente de alto nível.

Exemplo do botão e da lâmpada, onde ambas as classes Botao e Lampada são classes concretas:

```
1
2       public class Botao
3       {
4           private Lampada _lampada;
5
6           public void Acionar()
7           {
8               if (condicao)
9                   _lampada.Ligar();
10          }
      }
```

O design acima viola o DIP uma vez que Botao depende de uma classe concreta Lampada. Ou seja, Botao conhece detalhes de implementação ao invés de termos identificado uma abstração para o design.

Que abstração seria essa? Botao deve ser capaz de tratar alguma ação e ligar ou desligar algum dispositivo, seja ele qual for: uma lâmpada, um motor, um alarme, etc.

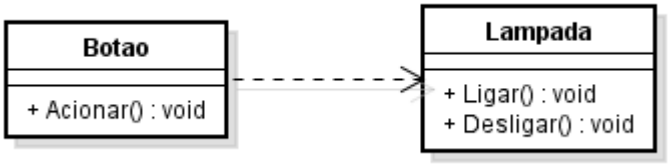
INVERTENDO A DEPENDÊNCIA:

A solução abaixo inverte a dependência de botão para a lâmpada, fazendo com que ambos agora dependam da abstração Dispositivo:

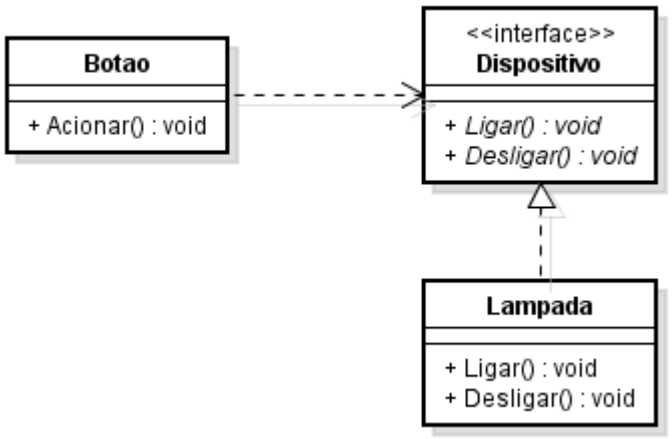
```
1       public class Botao
2       {
3           private Dispositivo _dispositivo;
4
5           public void Acionar()
6           {
7               if (condicao)
8                   _dispositivo.Ligar();
9           }
10      }
11
12      public interface Dispositivo
13      {
14          void Ligar();
15          void Desligar();
      }

      public class Lampada : Dispositivo
```

```
16      {
17          public void Ligar()
18          {
19              // ligar lampada
20          }
21          public void Desligar()
22          {
23              // desligar lampada
24          }
25      }
26
```

Ilustrando com UML, o que antes era:



Passa a ser:



Padrões de Projeto

1- O que são padrões de projeto

Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca a implementar duas vezes da mesma forma.

2- Quando utilizar

O alvo principal do uso dos padrões de projeto no desenvolvimento de software é o da orientação a objetos. Como os objetos são os elementos chaves em projetos OO, a parte mais difícil do projeto é a decomposição de um sistema em objetos. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização e assim por diante. Todos influenciam a decomposição, frequentemente de formas conflitantes.

3- Benefícios

O uso de padrões de projeto propicia a construção de aplicações e ou estruturas de código de forma flexível e a documentação de soluções reaproveitáveis. Através dos padrões de projeto é possível identificar os pontos comuns entre duas soluções diferentes para um mesmo problema. Conhecer esses pontos comuns nos permite desenvolver soluções cada vez melhores e mais eficientes que podem ser reutilizadas, permitindo, assim, o avanço do conhecimento humano.

Os padrões possibilitam através de uma linguagem clara e concisa, que os projetistas experientes transfiram os seus conhecimentos aos mais novos em um alto nível de abstração e assim facilitam o desenvolvimento e o reaproveitamento de código.

4- Tipos de Padrões de Projeto

1- Padrões de Criação (Creational)

Abstract Factory - Um método Factory é um método que fabrica objetos de um tipo particular; Um objeto Factory é um objeto que encapsula métodos Factory.

Builder - Separa a construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.

Factory Method - É uma interface para instanciação de objetos que mantém isoladas as classes concretas usadas da requisição da criação destes objetos.

Prototype - O padrão Prototype fornece uma outra maneira de se construir objetos de tipos arbitrários.

Singleton. - Garante que para uma classe específica só possa existir uma única instância, a qual é acessível de forma global e uniforme.

2- Padrões de Estrutura (Structural)

Adapter - Permite que dois objetos se comuniquem mesmo que tenham interfaces incompatíveis.

Bridge - Desacopla a interface da implementação ; Ocultação de detalhes de implementação dos clientes.

Composite - lida com uma estrutura de elementos agrupada hierarquicamente (não como meras coleções).

Decorator - Atribui responsabilidade adicionais a um objeto dinamicamente. O Decorator fornece uma alternativa flexível a subclasses para a extensão da funcionalidade.

Facade - Interface unificada para um subsistema ; Torna o subsistema mais fácil de usar.

Flyweight - Usa compartilhamento para dar suporte a vários objetos de forma eficiente.

Proxy - Fornece um objeto representante ou procurador de outro objeto para controlar o acesso ao mesmo.

3- Padrões de Comportamento (Behavioral)

Chain of Responsability - Evita dependência do remetente(cliente) de uma requisição ao seu destinatário , dando a oportunidade de mais de objeto tratar a requisição.

Command - Associa uma ação a diferentes objetos através de uma interface conhecida.

Interpreter - Usado para ajudar uma aplicação a entender uma declaração de linguagem natural e executar a funcionalidade da declaração.

Iterator - Provê uma forma de percorrermos os elementos de uma coleção sem violar o seu encapsulamento.

Mediator - Cria um objeto que age como um mediador controlando a interação entre um conjunto de objetos.

Memento - Torna possível salvar o estado de um objeto de modo que o mesmo possa ser restaurado.

Observer - Define uma relação de dependência 1:N de forma que quando um certo objeto (assunto) tem seu estado modificado os demais (observadores) são notificados; Possibilita baixo acoplamento entre os objetos observadores e o assunto.

State - Permite objeto alterar seu comportamento quando estado interno muda.

Strategy - Permite que uma família de algoritmos seja utilizada de modo independente e seletivo.

Template Method -Define o esqueleto de um algoritmo em uma operação adiando a definição de alguns passos para a subclasse.

Visitor - Define operações independentes a serem realizadas sobre elementos de uma estrutura.

Referencias

- http://www.devmedia.com.br/conceitos-interfaces-programacao-orientada-a-objetos-parte-1/18695
- http://pt.stackoverflow.com/questions/101552/o-que-%C3%A9-princ%C3%ADpio-da-invers%C3%A3o-de-depend%C3%AAncia-dip
- https://robsoncastilho.com.br/2013/05/01/principios-solid-principio-da-inversao-de-dependencia-dip/
- http://eduardopires.net.br/2013/04/orientacao-a-objeto-solid/
- https://robsoncastilho.com.br/2013/03/21/principios-solid-principio-de-substituicao-de-liskov-lsp/
- http://www.macoratti.net/vb_pd1.htm
- http://www.devmedia.com.br/conheca-os-padroes-de-projeto/957
- www.ime.usp.br/~kon/MAC5714/aulas/slides/GoF.ppt