SOLID Gubee Estudos

▼ **Single Responsability**

▼ Significado

O princípio de responsabilidade única é o primeiro princípio da sigla SOLID. “Uma classe deve ter apenas um motivo para mudar.” Cada módulo ou classe deve ter responsabilidade sobre uma única parte da funcionalidade fornecida pelo software, e essa responsabilidade deve ser totalmente encapsulada pela classe

▼ Exemplo de implementação

Aqui vemos que a classe CalcularaDeSalario alem de Calcular o salario ela tbm tem a responsabilidade de calcular com base no cargo no funcionario até ai ta tudo certo o problema começa caso a regra de calculo começa a crescer assim os ifs e elfs crescerão juntos Deixando a situação caotica e de dificil reciclagem

```
public class CalculadoraDeSalario {


    public double calcula(Funcionario funcionario) {
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return dezOuVintePorcento(funcionario);
        }

        if(DBA.equals(funcionario.getCargo()) || TESTER.equals(funcionario.getCargo())) {
            return quinzeOuVinteCincoPorcento(funcionario);
        }

        throw new RuntimeException("funcionario invalido");
    }

    private double dezOuVintePorcento(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 3000.0) {
            return funcionario.getSalarioBase() * 0.8;
        }
        else {
            return funcionario.getSalarioBase() * 0.9;
        }
    }

    private double quinzeOuVinteCincoPorcento(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 2000.0) {
            return funcionario.getSalarioBase() * 0.75;
        }
        else {
            return funcionario.getSalarioBase() * 0.85;
        }
    }
```

```
}
```

Para implementar o principio da responsabilidade unica é nessario deixar a classe menor e mais coesa separam a responsabilidade verificar os cargos criando uma interface para isso e criando classes para implementar essa interface exemplo da interface :

```
public interface RegraDeCalaculo {

  public Double calcula(EstudosGubeeFuncionario funcionario);

}
```

Classes que implementam ela para fazer o calculo:

```
public class DezPorcento implements RegraDeCalaculo {

    @Override
    public Double calcula(EstudosGubeeFuncionario funcionario) {
        return  funcionario.getSalario() * 0.1;
    }
}
```

```
public class VintePorcento implements RegraDeCalaculo {
    @Override
    public Double calcula(EstudosGubeeFuncionario funcionario) {
        return funcionario.getSalario() * 0.2;
    }
}
```

Nova Classe CalculadoraDeSalario:

```
public class CalculadoraDeSalario {

   public double calcula(Funcionario funcionario){
      return funcionario.calcularSalario();
   }

}
```

▼ **Open Closed**

▼ Significado

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

A moral da história é a seguinte: **quando eu precisar estender o comportamento de um código, eu crio código novo ao invés de alterar o código existente**.

▼ implementação

EXEMPLO DE VIOLAÇÃO DO PRINCIPIO:

```
public class Arquivo
{
}

public class ArquivoWord : Arquivo
{
    public void GerarDocX()
    {
        // codigo para geracao do arquivo
    }
}

public class ArquivoPdf : Arquivo
{
    public void GerarPdf()
    {
        // codigo para geracao do arquivo
    }
}

public class GeradorDeArquivos
{
   public void GerarArquivos(IList<Arquivo> arquivos)
   {
      foreach(var arquivo in arquivos)
      {
         if (arquivo is ArquivoWord)
            ((ArquivoWord)arquivo).GerarDocX();
         else if (arquivo is ArquivoPdf)
            ((ArquivoPdf)arquivo).GerarPdf();
      }
   }
}
```

o EXEMPLO ACIMA É BEM CLARO existe uma classe Geradora de arquivos que geram arquivos do tipo Word e pdf verificando se a arquivo é instanciado pela classe

ArquivoPdf ele gera um pdf mesma coisa com o word

o problema dessa implementação:

1) Alterar todos os métodos que precisem fazer uso do novo formato (certamente aqueles com vários “if/else if” ou um belo “switch..case”).

2) Recompilar e fazer o *deploy* de todos os componentes que foram impactados.

Quando uma mudança dessas acaba causando 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 dessas partes** do código.

Exemplo da implementação sem ferir o OCP:

```
public abstrac class Arquivo
{
  public void Gerar();
}

public class ArquivoWord : Arquivo
{
    public void Gerar()
    {
        // codigo para geracao do arquivo
    }
}

public class ArquivoPdf : Arquivo
{
      public void Gerar()
    {
        // codigo para geracao do arquivo
    }
}

public class GeradorDeArquivos
{
   public void GerarArquivos(IList<Arquivo> arquivos)
   {
      foreach(var arquivo in arquivos)
      {
         arquivo.gerar();
      }
   }
}
```

Agora, sempre que surgir um novo formato de arquivo, nós conseguimos estender o comportamento de “GerarArquivos” (ele saberá gerar esse novo arquivo) sem precisarmos alterá-lo. Apenas criamos o arquivo novo e pronto. Nada mais a fazer!

▼ **Subistituição de Liskov**

▼ Significado

O Princípio de Substituição de Liskov leva esse nome por ter sido criado por Barbara Liskov, em 1988. A definição formal de Liskov diz que:

“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”

Em outras palavras, toda e qualquer classe derivada deve poder ser usada como se fosse a classe base.

▼ Exemplo (Explicação)

Imagine que em um super Mercado existem dois tipos de clientes o vip é o normal onde o cliente normal é a classe base o cliente vip dev ser capaz de fazer tudo que cliente normal faz sem que haja implementação na sua classe

Uma exemplo muito comum da violação desse principio seria Quadrado é um Retangulo ?

onde o quadra na vida Real é Sim um retangulo Entre tanto ao Se um quadrado herdar os comportamentos do de um retangulo ele quebra o principio de liskov pois ele tera que fazer uma alteração pois o retangulo recebe altura e largura ja o quadrado esses valores são iguais

▼ **Segregação de Interface**

▼ Significado

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.

▼ implementação

Exemplo de violação:

```
interface pública MembroDeTimeScrum
{
void PriorizarBacklog ();
void BlindarTime ();
void ImplementarFuncionalidades ();
}

public class Dev : MembroDeTimeScrum
{
    public void PriorizarBacklog() { }
    public void BlindarTime() { }
```

```
    public void ImplementarFuncionalidades()
    {
        Console.Writeline("Codando e tomando café compulsivamente!!");
    }
}

public class ScrumMaster : MembroDeTimeScrum
{
    public void PriorizarBacklog() { }

    public void BlindarTime()
    {
        Console.Writeline("Devs working! You shall not pass!!!!");
    }

    public void ImplementarFuncionalidades() { }
}

public class ProductOwner : MembroDeTimeScrum
{
    public void PriorizarBacklog()
    {
        Console.Writeline("Priorizando backlog com base nas minhas necessidades de negócio");
    }

    public void BlindarTime() { }
    public void ImplementarFuncionalidades() { }
}
```

A interface de TimeScrum() é muito generica obrigando o dev a criar metodos que nao seria usado;

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.

Exemplo de implementação correta seria a criar de uma iterface mais especifaca para cada um

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

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

▼ **Inversão de Dependência**

▼ Significado

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;– Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Inverter a dependência faz com que um cliente não fique frágil a mudanças relacionadas a detalhes de implementação. Isto é, alterar o detalhe não quebra o cliente. Além disso, o mesmo cliente pode ser reutilizado com outro detalhe de implementação.

▼ Implementação

Exemplo de violação:

```
public class Botao
{
private Lampada _lampada;

 public void Acionar ()
 {
     se (condicao)
         _lampada.Ligar ();
 }


}
```

O DIP é violado no momento em que o Botão umas classe concreta Depende da lampada outra classe concreta em outras palavras Botao conhece detalhes de implementação ao invés de termos identificado uma abstração para o design ou seja o botão deve ser capaz de realizar uma ação de desligar ligar um dipositivo seja ele qual for: lamapda, computador, televisão dentre outros.

Exemplo de implementação Correta:

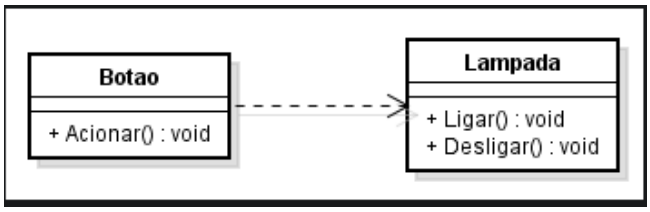
A solução a baixo inverte as depencias em vez do botão depender diretamente do lampada e a lampada do bitão ambos dependem da abstração do Disposito

```
public class Botao
{
    private Dispositivo _dispositivo;
```

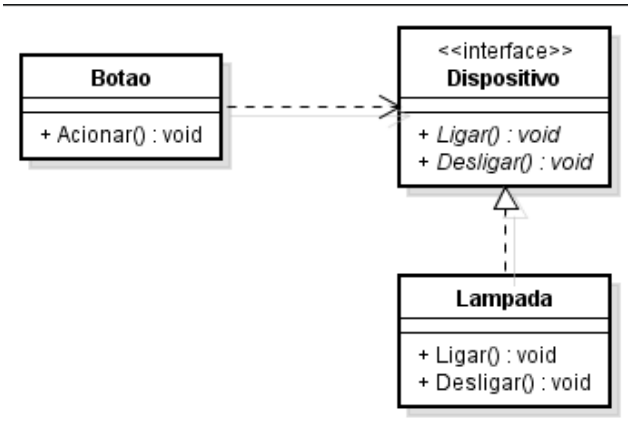
```
    public void Acionar()
    {
        if (condicao)
            _dispositivo.Ligar();
    }
}

public interface Dispositivo
{
    void Ligar();
    void Desligar();
}

public class Lampada : Dispositivo
{
    public void Ligar()
    {
        // ligar lampada
    }
    public void Desligar()
    {
        // desligar lampada
    }
}
```

Ilustração em ULM

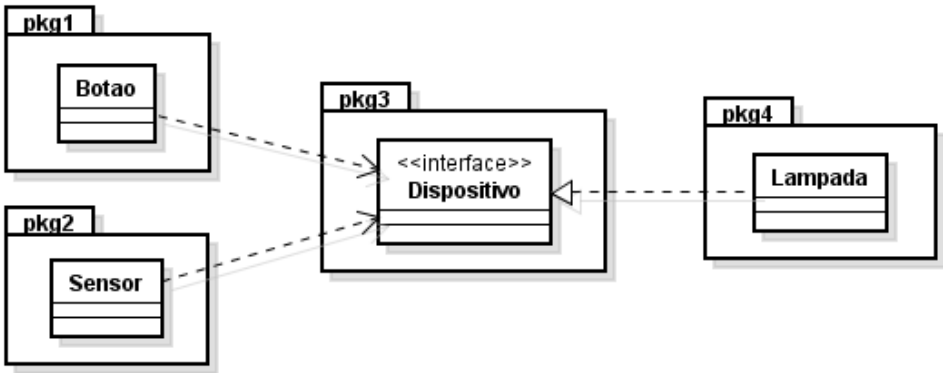
O que Era:



O que passou a ser:



Como seria caso tivesse outros dispositivos



CONCLUSÃO

O Princípio da Inversão de Dependência é um princípio essencial para um bom design orientado a objetos, ao passo que o oposto leva a um design engessado e procedural.

Identificar abstrações e inverter as dependências garantem que o software seja mais flexível e robusto, estando melhor preparado para mudanças.

Encerramos assim a série sobre os princípios SOLID, que juntos formam um conjunto de boas práticas que devemos ter em nosso cinto de utilidades e que devemos aplicar sempre que pudermos para melhorar a qualidade do design e da arquitetura do software.