



PRINCÍPIOS S.O.L.I.D





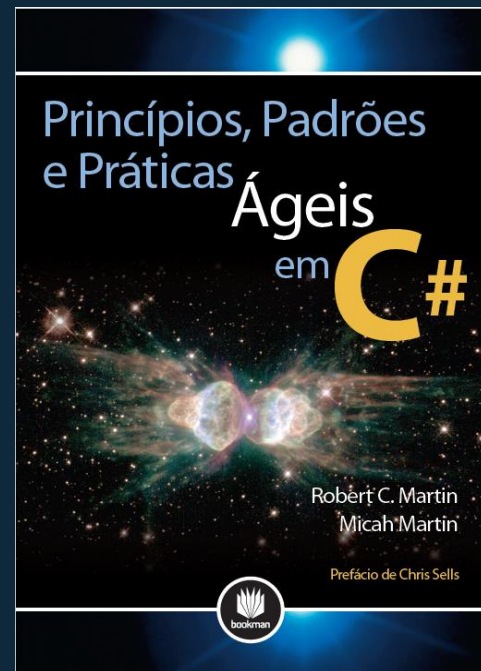
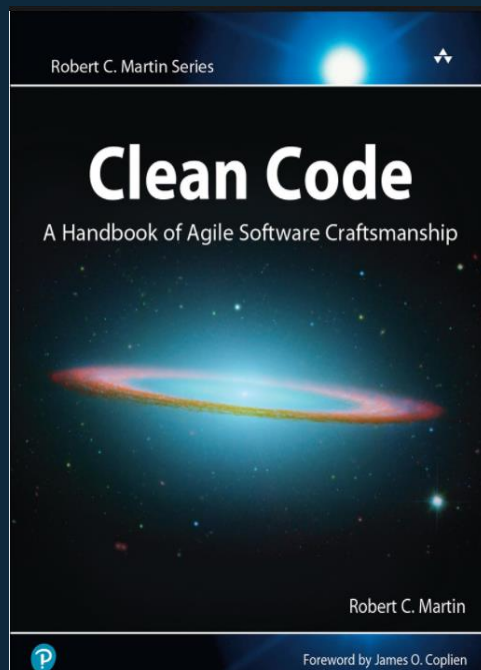
Olá!

Eu sou Willian Brito

- ❖ Desenvolvedor Full Stack na Msystem Software
- ❖ Formado em Analise e desenvolvimento de sistemas.
- ❖ Pós Graduado em Segurança Cibernética.
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018



Este conteúdo é baseado nestes livros





O QUE É O S.O.L.I.D ?

São princípios de como se deve projetar classes flexíveis na orientação a objetos. Esses princípios tem como objetivo ajudar o desenvolvedor a escrever códigos mais limpos, separando responsabilidades, diminuindo acoplamentos, facilitando na refatoração e evolução do software, estimulando o reaproveitamento do código.



Resumo do S.O.L.I.D

Letra	Sigla	Nome	Definição
S	SRP	Princípio da Responsabilidade Única	Uma classe deve ter um, e somente um, motivo para mudar.
O	OCP	Princípio do Aberto-Fechado	Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo.
L	LSP	Princípio da Substituição de Liskov	Objetos de sub classe podem ser substituídos por objetos de sua super classe.
I	ISP	Princípio da Segregação da Interface	Muitas interfaces específicas são melhores do que uma interface única.
D	DIP	Princípio da inversão da dependência	Dependa de uma abstração e não de uma implementação.



1

Single Responsibility Principle


"Uma classe deve ter um, e somente um, motivo para mudar."



Explicando o SRP

- ◇ Definição de Classe Coesa: Uma classe coesa é aquela que possui uma única responsabilidade.
- ◇ Esse princípio declara que uma classe deve ser coesa, tornando-a especializada em uma única regra de negócio e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.






Um exemplo de classe não coesa

```
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");  
    }  
}
```






Repare que cada uma das regras é implementada por um método privado, como o **dezOuVintePorcento()** e o **quinzeOuVinteCincoPorcento()**. Veja um exemplo da implementação desses métodos:

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





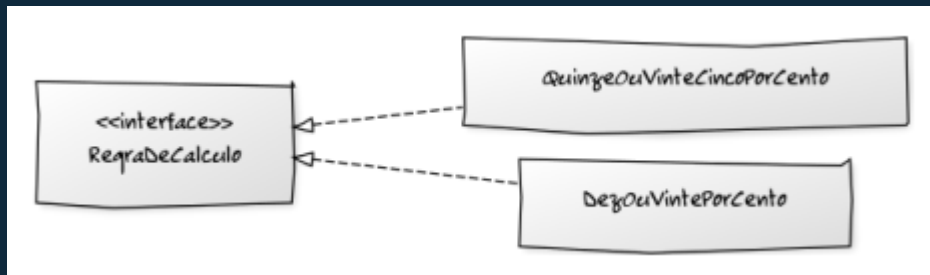
Qual o problema dessa classe?

- ◇ **Quantidade de cargos:** (desenvolvedor, DBA e tester) com regras similares, ou seja, caso tenha a necessidade de implementar mais cargos terá que acrescentar mais if e else.
- ◇ **Falta de Reuso:** A necessidade de se reutilizar o método **dezOuVintePorcento()** em algum outro ponto do sistema, terá que levar a classe **CalculadoraDeSalario** inteira para outro ponto do sistema, ou mesmo fazer outra classe depender dela só para reutilizar esse comportamento.



Em busca da coesão:

- ◇ A classe **CalculadoraDeSalario**, em particular, cresce indefinidamente por dois motivos: sempre que um cargo novo surgir ou sempre que uma regra de cálculo nova surgir. a ideia, portanto, será colocar cada uma dessas regras em classes diferentes, todas implementando a mesma interface.



Separando as Responsabilidades:

```
public class DezOuVintePorCento implements RegraDeCalculo {  
    public double calcula(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 3000.0) {  
            return funcionario.getSalarioBase() * 0.8;  
        }  
        else {  
            return funcionario.getSalarioBase() * 0.9;  
        }  
    }  
}  
  
public class QuinzeOuVinteCincoPorCento implements  
RegraDeCalculo {  
    public double calcula(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 2000.0) {  
            return funcionario.getSalarioBase() * 0.75;  
        }  
        else {  
            return funcionario.getSalarioBase() * 0.85;  
        }  
    }  
}
```


2

```
public interface RegraDeCalculo {  
    double calcula(Funcionario f);  
}
```

1

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


3



Como identificar se uma classe é coesa ?

É realmente difícil enxergar a responsabilidade de uma classe. Talvez essa seja a maior dúvida na hora de se pensar em códigos coesos. É fácil entender que a classe deve ter apenas uma responsabilidade. O difícil é definir o que é uma responsabilidade, afinal é algo totalmente subjetivo. Por isso colocamos mais código do que deveríamos nela. **Dois comportamentos “pertencem” ao mesmo conceito/ responsabilidade se ambos mudam juntos.**





Problemas com a violação do princípio

- ◇ Falta de coesão - uma classe não deve assumir responsabilidades que não são suas;
- ◇ Alto acoplamento - Mais responsabilidades geram um maior nível de dependências, deixando o sistema engessado e frágil para alterações;
- ◇ Dificuldades na implementação de testes automatizados - É difícil de “mockar” esse tipo de classe;
- ◇ Dificuldades para reaproveitar o código;





Vantagens / Desvantagens


Vantagens

- ◇ Melhor manutenibilidade
- ◇ Melhor testabilidade
- ◇ Maior reuso de código
- ◇ Classes pequenas e mais coesas

Desvantagens

- ◇ Mais arquivos para manipular
- ◇ Mais complexidade no código





Padrões de projeto que auxiliam na implementação do SRP

- ◇ Chain of Responsibility
- ◇ State
- ◇ Decorator





Conclusão do SRP

Classes coesas são mais fáceis de serem mantidas, reutilizadas e tendem a ter menos bugs. Pense nisso, coesão é fundamental.



A decorative graphic on the left side of the slide. It features a large cyan hexagon with the number '2' inside. Surrounding this central hexagon are several smaller hexagons of varying shades of blue and cyan. Some of these smaller hexagons contain white icons: a lightbulb, a thumbs-up, a smartphone, a magnifying glass, and a gear. There is also a network-like icon with a central node and connecting lines.

2

Open/Closed Principle

Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação.



Explicando o OCP

- ◇ A ideia é que suas classes sejam abertas para extensão. Ou seja, estender o comportamento delas deve ser fácil. Mas, ao mesmo tempo, elas devem ser fechadas para alteração. Ou seja, ela não deve ser modificada (ter seu código alterado) o tempo todo.



Exemplo de Violação do OCP:


◇ O código é bem simples. Ele basicamente pega um produto da loja e tenta descobrir seu preço. Ele primeiro pega o preço bruto do produto, e aí usa a tabela de preços padrão (**TabelaDePrecoPadrao**) para calcular o preço; pode-se ter um eventual desconto. Em seguida, o código descobre também o valor do frete. Por fim, ele faz a conta final: valor do produto, menos desconto, mais o frete.

```
public class CalculadoraDePrecos {  
  
    public double calcula(Compra produto) {  
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();  
        Frete correios = new Frete();  
  
        double desconto =  
            tabela.descontoPara(produto.getValor());  
        double frete = correios.para(produto.getCidade());  
  
        return produto.getValor() * (1-desconto) + frete;  
    }  
}
```

Exemplo de Violação do OCP:

◇ Veja que, nesse exemplo, se quisermos mudar a maneira com que o cálculo de frete é feito, precisamos por as mãos nessa classe. Como possibilitar que a regra de frete seja alterada sem a necessidade de mexer nesse código? O primeiro passo é criarmos uma abstração para o problema, e fazer com que essas abstrações possam ser injetadas na classe que as usa. Se temos diferentes regras de desconto e de frete, basta criarmos interfaces que as representam:

```
public class TabelaDePrecoPadrao {  
    public double descontoPara(double valor) {  
        if(valor>5000) return 0.03;  
        if(valor>1000) return 0.05;  
        return 0;  
    }  
}  
  
public class Frete {  
    public double para(String cidade) {  
        if("SAO PAULO".equals(cidade.toUpperCase())) {  
            return 15;  
        }  
        return 30;  
    }  
}
```



Qual o problema dessa classe?

◇ Imagine que o sistema é mais complicado que isso. Não existe apenas uma única regra de cálculo de desconto, mas várias; e também não existe apenas uma única regra de frete, existem várias. Uma maneira (infelizmente) comum de vermos código por aí é resolvendo isso por meio de ifs. Ou seja, o código decide se é a regra A ou B que deve ser executada. O código a seguir exemplifica os ifs para diferentes tabelas de preços:



```
public class CalculadoraDePrecos {

    public double calcula(Compra produto) {

        Frete correios = new Frete();

        double desconto;
        if (REGRA 1){
            TabelaDePrecoPadrao tabela =
                new TabelaDePrecoPadrao();
            desconto = tabela.descontoPara(produto.getValor());
        }
        if (REGRA 2){
            TabelaDePrecoDiferenciada tabela =
                new TabelaDePrecoDiferenciada();
            desconto = tabela.descontoPara(produto.getValor());
        }
        double frete = correios.para(produto.getCidade());
        return produto.getValor() * (1 - desconto) + frete;
    }
}
```

Classes Flexíveis

◇ Perceba que a classe agora está aberta para extensão. Afinal, basta passarmos diferentes implementações de tabela e de frete para que ela execute de maneira distinta. Ao mesmo tempo, está fechada para modificação, afinal não há razões para mudarmos o código dessa classe. Essa classe agora segue o princípio do aberto-fechado.

```
public interface TabelaDePreco {
    double descontoPara(double valor);
}

public class TabelaDePreco1 implements TabelaDePreco { }
public class TabelaDePreco2 implements TabelaDePreco { }
public class TabelaDePreco3 implements TabelaDePreco { }

public interface ServicoDeEntrega {
    double para(String cidade);
}

public class Frete1 implements ServicoDeEntrega {}
public class Frete2 implements ServicoDeEntrega {}
public class Frete3 implements ServicoDeEntrega {}
```




Classes Flexíveis

◇ Além disso, já que temos diferentes implementações, é necessário também que a troca entre elas seja fácil. Para isso, a solução é deixar de instanciar as implementações concretas dentro dessa classe, e passar a recebê-las pelo construtor.

```
public class CalculadoraDePrecos {  
  
    private TabelaDePreco tabela;  
    private ServicoDeEntrega entrega;  
  
    public CalculadoraDePrecos(  
  
        TabelaDePreco tabela,  
        ServicoDeEntrega entrega) {  
  
        this.tabela = tabela;  
        this.entrega = entrega;  
    }  
}
```





Classes Flexíveis

◇ Sempre que instanciamos classes diretamente dentro de outras classes, perdemos a oportunidade de trocar essa implementação em tempo de execução. Ou seja, se instanciamos **TabelaDePreco1** diretamente no código da classe principal, será sempre essa implementação concreta que será executada. E não queremos isso, queremos conseguir trocar a tabela de preço quando quisermos.

```
// não instanciamos mais as dependências aqui,  
// apenas as usamos.  
public double calcula(Compra produto) {  
    double desconto =  
        tabela.descontoPara(produto.getValor());  
    double frete = entrega.para(produto.getCidade());  
  
    return produto.getValor() * (1-desconto) + frete;  
}
```





Classes Flexíveis

◇ Portanto, em vez de instanciarmos as classes de maneira fixa, vamos recebê-las por construtores. Veja que essa simples mudança altera toda a maneira de se lidar com a classe. Com ela “aberta”, ou seja, recebendo as dependências pelo construtor, podemos passar a implementação concreta que quisermos para ela. Se passarmos a implementação **TabelaDePreco1**, e invocarmos o método **calcula()**, o resultado será um; se passarmos a implementação **TabelaDePreco2** e invocarmos o mesmo método, o resultado será outro. Ou seja, conseguimos mudar o comportamento final da classe **CalculadoraDePrecos** sem mudar o seu código. Como conseguimos isso? Justamente porque ela está aberta. É fácil mudar o seu comportamento interno, porque ela depende de abstrações e nos possibilita mudar essas dependências a qualquer momento.





Vantagens / Desvantagens


Vantagens

- ◇ A principal vantagem é a facilidade na adição de novos requisitos, diminuindo as chances de introduzir novos bugs, pois o novo comportamento fica isolado, e o que estava funcionando provavelmente continuara funcionando.
- ◇ Flexibilidade

Desvantagens

- ◇ Mais arquivos para manipular
- ◇ Mais complexidade no código





Padrões de projeto que auxiliam na implementação do OCP

- ◇ Strategy
- ◇ Template Method
- ◇ Decorator





Conclusão do OCP

Classes abertas são aquelas que deixam explícitas as suas dependências. Dessa maneira, podemos mudar as implementações concretas que são passadas para ela a qualquer momento, e isso faz com que o resultado final da sua execução mude de acordo com as classes que foram passadas para ela. Ou seja, conseguimos mudar o comportamento da classe sem mudar o seu código. Lembre-se que sistemas OO evoluem por meio de novos códigos, e não de alterações em códigos já existentes. Programar OO é um desafio. Mas um desafio divertido.



A decorative pattern of hexagons in various shades of blue and teal. Some hexagons contain icons: a lightbulb, a thumbs up, a smartphone, a magnifying glass, and a gear. A large hexagon in the center-left contains the number 3.

3

Liskov Substitution Principle

“Se 'S' é um subtipo de T, então os objetos do tipo 'T', podem ser substituídos por objetos do tipo 'S' sem que essa substituição gere efeitos colaterais na nossa aplicação.”



Explicando o LSP

- ◇ Objetos de sub classe podem ser substituídos por objetos de sua super classe.
- ◇ Em outras palavras, **toda e qualquer classe derivada deve poder ser usada como se fosse a classe base**, ou seja na prática o principio de Liskov tem como objetivo nos ensinar a ESTRUTURAR muito bem CLASSES ABSTRATAS e implementar HERANÇA e POLIMORFISMO da forma correta.





Herança e Polimorfismo

- ◇ Herança é sempre um assunto delicado. No começo das linguagens orientadas a objeto, a herança era a funcionalidade usada para vender a ideia. Afinal, reuso de código de maneira fácil, quem não queria? Mas, na prática, utilizar herança pode não ser tão simples. É fácil cair em armadilhas criadas por hierarquias de classes longas ou confusas.





Herança e Polimorfismo

```
public class ContaComum {  
  
    protected double saldo;  
  
    public ContaComum() {  
        this.saldo = 0;  
    }  
  
    public void deposita(double valor) {  
        if(valor <= 0)  
            throw new ValorInvalidoException();  
  
        this.saldo += valor;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}  
  
    public void rende() {  
        this.saldo*= 1.1;  
    }  
}
```



Herança e Polimorfismo

◇ A classe **ContaComum** representa, de maneira simplificada, uma conta em um banco. A classe possui operações simples como **deposita()** e **rende()**. Mas, como sempre, o sistema precisa crescer. Imagine agora a classe **ContaDeEstudante**, que é exatamente igual a uma conta, com a diferença de que ela não “rende”. Usando herança, a implementação seria algo parecido com a que segue, onde o método **rende()** lança uma exceção:

```
public class ContaDeEstudante extends ContaComum {  
  
    public void rende() {  
        throw new ContaNaoRendeException();  
    }  
}
```

Qual o problema dela ?

- ◇ É difícil enxergar o problema dessa simples sobrescrita. Para isso, imagine um código que faz uso de ambas **ContaComum** e **ContaDeEstudante**:
- ◇ O método **contasDoBanco()** retorna uma lista com diferentes contas. Não sabemos exatamente quais estão lá dentro, mas, dado o polimorfismo, podemos tratar todas elas pela referência da classe pai.

```
public class ProcessadorDeInvestimentos {  
  
    public static void main(String[] args) {  
  
        for (ContaComum conta : contasDoBanco()) {  
            conta.rende();  
  
            System.out.println("Novo Saldo:");  
            System.out.println(conta.getSaldo());  
        }  
    }  
}
```



Qual o problema dela ?

◇ **Agora o problema:** qual o comportamento da aplicação? Não sabemos. Afinal, se houver alguma conta de estudante nesse código, a execução do programa parará, pois uma exceção será lançada. Pense que o sistema possui diversos desses loops e classes que interagem com **ContaComum** (e, por consequência, interagem com qualquer filho dela também). A nova classe **ContaDeEstudante** pode fazer essas classes pararem de funcionar também. Por que isso aconteceu? Porque a classe filha quebrou o contrato definido pela classe pai: o método **rende()** na classe pai não lança exceção. Ou seja, as classes clientes não vão esperar que isso aconteça, e não vão tratar essa possibilidade. Note que as classes filhas precisam respeitar os contratos definidos pela classe pai. Mudar esses contratos pode ser perigoso.





Aplicando o LSP

◇ Para usar polimorfismo de maneira adequada, o desenvolvedor deve respeitar as pré-condições **Contravariância** (\geq) e pós-condições **Covariância** (\leq) que a classe pai definiu. Toda classe ou método tem as suas pré e pós-condições.

- ◇ **Invariância:** propriedade, condição ou atributo que nunca sofre variação, ou seja sempre permanece no estado de verdadeiro durante a vida do objeto, para que o estado do objeto seja considerado válido.
- ◇ **Pré-condições ou Contravariância** (\geq): É quando os dados que chegam nela possui restrições iniciais para que aquele método funcione corretamente, esses parâmetros de **entrada**, não podem ser mais restritos em uma sub-classe. Isto significa que só é permitida uma sub-classe com números de possibilidades válidas \geq ou **tipos de objetos mais genéricos** que a sua super-classe.
- ◇ **Pós-condições ou Covariância** (\leq): São o outro lado da moeda. Ou seja, Qual a **saída** ou o **retorno** do comportamento do método, esta saída não pode ser ampliada em uma sub-classe. Isto significa que só é permitida uma sub-classe que contém um numero de possibilidades de retorno ou saída válidas \leq ou **tipos de objetos mais especializados** que a sua super-classe.



Aplicando o LSP

◇ Exemplo de **Pré-condição ou Contravariância (\geq)**, o método **deposita()** deve receber um inteiro maior que zero. Ou seja, o valor “1” é válido. Se uma classe filha de **ContaComum** mudar essa pré-condição para somente números maiores que 10, poderemos ter problemas, pois na super classe os números de 1 a 10 são válidos e na sub classe estamos diminuindo a possibilidade de números válidos, ou seja quebramos o contrato com a super classe.

```
public void deposita(double valor) {  
    if(valor <= 0)  
        throw new ValorInvalidoException();  
  
    this.saldo += valor;  
}
```



Aplicando o LSP

◇ Exemplo de **Pós-condição ou Covariância (\leq)**, O método **rende()** não devolve nada e não lança exceção. No exemplo que demos, já a classe **ContaDeEstudante**, esse mesmo método lança uma exceção.

```
public void rende()    {  
    this.saldo*= 1.1;  
}
```

```
public class ContaDeEstudante extends ContaComum {  
  
    public void rende() {  
        throw new ContaNaoRendeException();  
    }  
}
```





Aplicando a Pré-condição / Contravariância (\geq)

◇ Podemos sim mudar as pré e pós-condições, mas com regras. A classe filho só pode afrouxar a precondição. Pense no caso em que a classe pai chamada **CalculadoraPrazo** tem um método que recebe inteiros relacionados aos dias do prazo, esta classe possui uma condição onde se os dias for maior que 0 ele retorna a data de hoje somada com os dias, caso o contrário retorna uma exceção.

```
class CalculadoraPrazo {  
    public function data(int $dias): DateTimeInterface {  
        if($dias > 0) {  
            return (new DateTime())->modify("+$dias days");  
        }  
        throw new \InvalidArgumentException("Prazo precisa ser maior que zero ");  
    }  
}
```





Aplicando a Pré-condição / Contravariância (\geq)

◇ Agora para respeitar o LSP vamos criar uma classe chamada **CalculadoraCLT** que herda da classe **CalculadoraPrazo**, e para respeitar a **Pré-condição / Contravariância (\geq)**. Iremos ampliar as possibilidades de valores válidos aplicando a condição se a quantidade de dias for \geq a 0 será retornado a data de hoje mais a quantidade de dias passado pelo parâmetro.

```
class CalculadoraPrazoCLT extends CalculadoraPrazo {  
    public function data(int $dias): \DateTimeInterface {  
        if($dias >= 0) {  
            return (new \DateTime())->modify("+ $dias days");  
        }  
        throw new \InvalidArgumentException("Prazo precisa ser maior ou igual a zero ");  
    }  
}
```





Violando a Pré-condição / Contravariância (\geq)

◇ Agora iremos violar a **Pré-condição / Contravariância (\geq)**, vamos criar uma classe chamada **CalculadoraPrazoCPC**, herdando de **CalculadoraPrazo** e iremos diminuir o numero de possibilidades válidas aplicando uma condição que irá verificar se a quantidade de dias está em um range de 1 á 30 se a condição for respeitada será retornado a data de hoje mais a quantidade de dias passado pelo parâmetro.

```
class CalculadoraPrazoCPC extends CalculadoraPrazo {  
    public function data(int $dias): \DateTimeInterface {  
        if(in_array($dias, range(1, 30))) {  
            return (new DateTime())->modify("+ $dias days");  
        }  
        throw new \InvalidArgumentException("Prazo precisa ser entre 1 e 30 ");  
    }  
}
```





Pré Condições Contravariância (\geq)

```
class Alimento {} // Objeto Genérico
class AlimentoAnimal extends Alimento {} // Objeto Especialista

abstract class Animal {
    protected string $nome;

    public function __construct(string $nome) {
        $this->nome = $nome;
    }

    public function comer(AlimentoAnimal $alimento) { // Entrada: Objeto Especialista
        echo $this->nome . "nhac nhac comendo" . get_class($alimento);
    }
}

class Gato extends Animal {}
class Cachorro extends Animal {
    public function comer(Alimento $alimento) { // Entrada: Objeto Genérico
        echo $this->nome . "chop chop comendo" . get_class($alimento);
    }
}
```

Resultados dos testes

```
$dias = 31;  
$calculadora = new CalculadoraPrazo();  
echo $calculadora->data($dias)->format('d/m/Y');
```



```
$dias = 31;  
$calculadora = new CalculadoraPrazoCLT();  
echo $calculadora->data($dias)->format('d/m/Y');
```



```
$dias = 31;  
$calculadora = new CalculadoraPrazoCPC();  
echo $calculadora->data($dias)->format('d/m/Y');
```




(!) Fatal error: Uncaught InvalidArgumentException: Prazo precisa ser entre 1 e 30 in C:\wamp64\www\curso\aula24.php on line 24

(!) InvalidArgumentException: Prazo precisa ser entre 1 e 30 in C:\wamp64\www\curso\aula24.php on line 24

Call Stack

#	Time	Memory	Function	Location
1	0.0007	363512	{main}()	...\aula24.php:0
2	0.0007	363552	CalculadorPrazoCPC->data()	...\aula24.php:31




Aplicando a Pós-condição / Covariância (\leq)

◇ Neste exemplo de pós condição temos uma classe chamada **ContaBancaria**, que recebe um saldo inicial passado pelo construtor, nela existe um método **sacar()**, que recebe como parâmetro o valor do saque e se o saldo menos o valor do saque \geq a 0 ele subtrai o valor do saque em cima do saldo da conta, e retorna o saldo, repare que essa condição é para garantir que o retorno sempre seja um valor positivo.

```
class ContaBancaria {  
    protected float $saldo;  
  
    public function __construct(float $saldoInicial) {  
        $this->saldo = $saldoInicial;  
    }  
  
    public function sacar(float $valor): float {  
        if($this->saldo - $valor >= 0)  
            $this->saldo -= $valor;  
        return $this->saldo;  
    }  
}
```






Aplicando a Pós-condição / Covariância (\leq)

◇ Agora vamos criar uma classe chamada **ContaBancariaVip** herdando de **ContaBancaria**, no método **sacar()**, existe uma constante que tem o valor da taxa do banco e para manter o princípio é feito uma condição que se o saldo menos o valor for \geq que taxa que o banco cobra nesta conta vip e a ideia é nunca deixar um valor mínimo da taxa que o banco cobra para manter esta conta, cumprindo com esta condição aí é permitido sacar o dinheiro da conta.

```
class ContaBancariaVip extends ContaBancaria {  
    private const TAXA = 10.00;  
  
    public function sacar(float $valor): float {  
        if( ($this->saldo - $valor) >= self::TAXA)  
            $this->saldo -= $valor;  
        return $this->saldo;  
    }  
}
```





Violando a Pós-condição / Covariância (\leq)

◇ Agora iremos violar a **Pós-condição / Covariância (\leq)**, vamos criar uma classe chamada **ContaBancariaIlimitada**, herdando de **ContaBancaria**, e na sobrescrita do método **sacar()** não temos condições de entrada ele simplesmente subtrai o valor do saque em cima do saldo da conta e retorna o saldo.

```
class ContaBancariaIlimitada extends ContaBancaria {  
    public function sacar(float $valor): float {  
        $this->saldo -= $valor;  
        return $this->saldo;  
    }  
}
```



Resultados dos testes

```
$saldoInicial = 100.00;  
$conta = new ContaBancaria($saldoInicial);  
$conta->sacar(80.00);  
echo $conta->sacar(21.00); // Saldo: R$ 20,00
```



```
$saldoInicial = 100.00;  
$conta = new ContaBancariaVip($saldoInicial);  
$conta->sacar(80.00);  
echo $conta->sacar(13.00); // Saldo: R$ 20,00
```



```
$saldoInicial = 100.00;  
$conta = new ContaBancariaIlimitada($saldoInicial);  
$conta->sacar(80.00);  
echo $conta->sacar(150.00); // Saldo: R$ -130,00
```



Pós Condições Covariância (\leq)

```
abstract class Animal {
    protected string $nome;

    public function __construct(string $nome) {
        $this->nome = $nome;
    }

    abstract public function fazerSom();
}

class Cachorro extends Animal {
    public function fazerSom() {
        echo $this->nome . ": au au !";
    }
}

class Gato extends Animal {
    public function fazerSom() {
        echo $this->nome . ": miau !";
    }
}
```

Pós Condições Covariância (<=)

```
interface AbrigoAnimal {  
    public function adotar(string $nome): Animal; //Saída: Objeto Genérico  
}
```

```
class AbrigoGato implements AbrigoAnimal {  
    public function adotar(string $nome): Gato {  
        return new Gato($nome);  
    }  
}
```

```
class CachorroShelter implements AbrigoAnimal {  
    public function adotar(string $nome): Cachorro {  
        return new Cachorro($nome); //Saída: Objeto Especialista  
    }  
}
```

```
$gatinho = (new AbrigoGato)->adotar("Darwin");  
$gatinho->fazerSom(); // Retorno: Darwin: miau !  
echo "<br />";  
$doguinho = (new CachorroShelter)->adotar("Bolota");  
$doguinho->fazerSom(); // Retorno: Bolota: au au !
```

Aplicando Invariância

◇ Neste exemplo de **invariância** temos uma classe chamada **Pessoa**, que contém os atributos **nome** e **apelido** e implementamos um método chamado **invariant()** que tem como regra o nome e o apelido deverão ser diferentes.

```
class Pessoa {  
    protected string $nome;  
    protected string $apelido;  
  
    function __construct(string $nome, string $apelido) {  
        $this->nome = $nome;  
        $this->apelido = $apelido;  
        $this->invariant();  
    }  
  
    public function __set($nome, $value){  
        $this->$nome = $value;  
        $this->invariant();  
    }  
  
    public function __get($nome){  
        $this->invariant();  
        return $this->$nome;  
    }  
  
    public function __set($apelido, $value){  
        $this->$apelido = $value;  
        $this->invariant();  
    }  
  
    public function __get($apelido){  
        $this->invariant();  
        return $this->$apelido;  
    }  
  
    protected function invariant() {  
        assert(($this->nome != $this->apelido));  
    }  
}
```



Aplicando Invariância

◇ Agora iremos criar uma classe chamada **BoaPessoa** herdando de **Pessoa** que possui apenas o método de **getNomeCompleto()**, podemos perceber que essa classe não sobrescreve o método **invariant()** da classe Pessoa e com isso está classe respeita o conceito de **invariância**.

```
class BoaPessoa extends Pessoa {  
    public function getNomeCompleto() {  
        return $this->nome." ".$this->apelido. "!!!";  
    }  
}
```





Violando Invariância

- ◇ Agora iremos criar uma classe chamada **MalvadaPessoa** herdando de **Pessoa** que possui apenas o método que sobrescreve o método **invariant()** da classe **Pessoa** aplicando a regra em que o nome deve ser diferente de vazio.
- ◇ Podemos perceber que ao aplicar essa regra estamos violando a **invariância**, pois a classe **MalvadaPessoa** permite que o nome e apelido sejam iguais.

```
class MalvadaPessoa extends Pessoa {  
    public function invariant() {  
        assert(($this->nome != ""));  
    }  
}
```



Resultados dos testes

```
$pessoa = new Pessoa("Hinata", "Hinatinha");  
$pessoa->apelido = "Hinata"; // Gera Exceção  
| | | | | | | // Não Permite que nome e apelido sejam iguais
```




```
$pessoa = new BoaPessoa("Hinata", "Hinatinha");  
$pessoa->apelido = "Hinata"; // Gera Exceção  
| | | | | | | // Não Permite que nome e apelido sejam iguais
```



```
$pessoa = new MalvadaPessoa("Hinata", "Hinatinha");  
$pessoa->apelido = "Hinata"; // Não Gera Exceção  
| | | | | | | // Permite que nome e apelido sejam iguais
```






Problemas com a violação do principio ?

- ◇ Lançar uma exceção inesperada;
- ◇ Retornar valores de tipos diferentes da classe base;
- ◇ Introduzir comportamentos inesperados





Padrões de projeto que auxiliam na implementação do LSP

- ◇ Composite
- ◇ Strategy
- ◇ Template Method





Conclusão do LSP

O princípio da Substituição de Liskov veio para garantir que não criaremos nenhum design estranho enquanto usamos a herança e polimorfismo. A herança e Polimorfismo é um mecanismo muito poderoso da Orientação a Objetos, mas potencialmente pode criar muita confusão e comportamentos estranhos e difíceis de serem detectados. Seguindo esse princípio os comportamentos ficam mais previsíveis ao longo da cadeia de heranças.



A decorative pattern of hexagons in various shades of blue and cyan. Some hexagons contain icons: a lightbulb, a thumbs up, a smartphone, a magnifying glass, a gear, and a speech bubble. A large cyan hexagon with the number '4' is the central focus of the pattern.

4

Interface Segregation Principle

Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.



Explicando o ISP

- ◇ Esse princípio basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica, ou seja separar os métodos mais especializados.
- ◇ É melhor ter várias interfaces curtas do que ter poucas interfaces longas, também conhecidas como interface monolítica, basicamente é o mesmo conceito que o principio SRP, porém para abstrações.



Exemplo de Violação do ISP

- ◇ Imagine a implementação dessa calculadora, para o imposto ISS, na qual o valor é 10% do valor cheio e a nota fiscal é gerada com os dados desse imposto.

```
interface Imposto {  
    NotaFiscal geraNota();  
    double imposto(double valorCheio);  
}  
  
class ISS implements Imposto {  
    public double imposto(double valorCheio) {  
        return 0.1 * valorCheio;  
    }  
  
    public NotaFiscal geraNota() {  
        return new NotaFiscal(  
            "Alguma informacao aqui",  
            "Alguma outra informacao aqui"  
        );  
    }  
}
```



Exemplo de Violação do ISP

- ◇ Agora imagine um novo imposto, chamado **IXMX**, que é calculado também sobre o valor cheio, mas não emite nota fiscal. Como implementar a classe concreta? O que fazer com o método **geraNota()**? Podemos lançar uma exceção ou retornar um valor nulo, por exemplo:

```
class IXMX implements Imposto {  
    public double imposto(double valorCheio) {  
        return 0.2 * valorCheio;  
    }  
  
    public NotaFiscal geraNota() {  
        // lança uma exceção  
        throw new NaoGeraNotaException();  
        // ou retornar nulo  
        return null;  
    }  
}
```




Aplicando o ISP

- ◇ A solução para o problema é análoga ao que tomamos quando discutimos classes coesas. Se uma classe não é coesa, dividimo-la em duas ou mais classes;
- ◇ Se uma interface não é coesa, também a dividimos em duas ou mais interfaces. Veja:

```
interface CalculadorDeImposto {  
    double imposto(double valorCheio);  
}  
  
interface GeradorDeNota {  
    NotaFiscal geraNota();  
}
```

```
class ISS implements CalculadorDeImposto, GeradorDeNota {  
    // os dois métodos aqui  
}  
  
class IXMX implements CalculadorDeImposto {  
    // só implementa uma interface, pois  
    // esse aqui não gera nota fiscal  
}
```



Problemas com a violação do princípio

- ◇ Falta de coesão - uma interface não deve assumir responsabilidades que não são suas;
- ◇ Dificuldades para reaproveitar a interface;





Vantagens / Desvantagens

Vantagens

- ◇ Maior reuso das interfaces
- ◇ Interfaces pequenas e mais coesas
- ◇ Criar abstrações de forma correta.

Desvantagens

- ◇ Mais arquivos para manipular
- ◇ Mais complexidade no código





Conclusão do ISP

Interfaces são fundamentais em bons sistemas orientados a objetos. Tomar conta delas é importante. Neste princípio, discutimos as chamadas interfaces gordas, que são aquelas interfaces que contem muitas responsabilidades diferentes. Assim como nas classes não coesas, essas interfaces também possuem baixo reuso e dificultam a manutenção. Agora você percebeu que acoplamento, coesão, simplicidade fazem sentido não só para classes concretas, mas sim para tudo.



A decorative pattern of hexagons in various shades of blue and cyan on the left side of the slide. Some hexagons contain icons: a lightbulb, a thumbs up, a network of nodes, a smartphone, a magnifying glass, a gear, and a speech bubble.

5

Dependency Inversion Principle

Dependa de abstrações e não de implementações.



Definição de Acoplamento: “Sempre que uma classe depende da outra para existir, é acoplamento.”



Exemplo de Violação do DIP


- ◇ A classe **GeradorDeNotaFiscal** é acoplada ao **EnviadorDeEmail** e **NotaFiscalDao**.

```
public class GeradorDeNotaFiscal {  
    private final EnviadorDeEmail email;  
    private final NotaFiscalDao dao;  
  
    public GeradorDeNotaFiscal(EnviadorDeEmail email,  
        NotaFiscalDao dao) {  
        this.email = email;  
        this.dao = dao;  
    }  
}
```



Exemplo de Violação do DIP


```
public NotaFiscal gera(Fatura fatura) {  
  
    double valor = fatura.getValorMensal();  
  
    NotaFiscal nf = new NotaFiscal(  
        valor,  
        impostoSimplesSobre0(valor)  
    );  
  
    email.enviaEmail(nf);  
    dao.persiste(nf);  
  
    return nf;  
}  
  
private double impostoSimplesSobre0(double valor) {  
    return valor * 0.06;  
}  
}
```



Qual o problema dessa classe?

- ◇ Pense agora o seguinte: hoje, esse código em particular manda e-mail e salva no banco de dados usando um **DAO**. Imagine que amanhã esse mesmo trecho de código também mandará informações para o **SAP**, disparará um **SMS**, consumirá um outro sistema da empresa etc. A classe **GeradorDeNotaFiscal** vai crescer, e passar a depender de muitas outras classes.






Qual o problema dessa classe?

- ◇ O grande problema do acoplamento é que uma mudança em qualquer uma das classes pode impactar em mudanças na classe principal. Ou seja, se o **EnviadorDeEmail** parar de funcionar, o problema será propagado para o **GeradorDeNotaFiscal**. Se o **NFDao** parar de funcionar, o problema será propagado para o gerador. E assim por diante.
- ◇ Portanto, o problema é a partir do momento em que uma classe possui muitas dependências, todas elas podem propagar problemas para a classe principal.






Buscando acoplamentos adequados

- ◇ Agora a próxima pergunta é: **será que conseguimos acabar com o acoplamento?** Ou seja, fazer com que as classes não dependam de nenhuma outra? E impossível. Nos sabemos que, na prática, quando estamos fazendo sistemas de médio/grande porte, as dependências existirão. O acoplamento vai existir. Uma classe dependerá de outra que, por sua vez, dependerá de outra, e assim por diante.
- ◇ Já que não é possível eliminar os acoplamentos, é necessário diferencia-los. Afinal, será que todo acoplamento é problemático igual? Ou será que alguns são menos piores que outros? Porque, caso isso seja possível, modelaremos nossos sistemas fugindo dos “acoplamentos perigosos”. O ponto chave é eliminar acoplamentos perigosos e buscar “acoplamentos adequados”
- ◇ Por mais estranho que pareça, é comum nos acoplarmos com classes e nem percebermos.





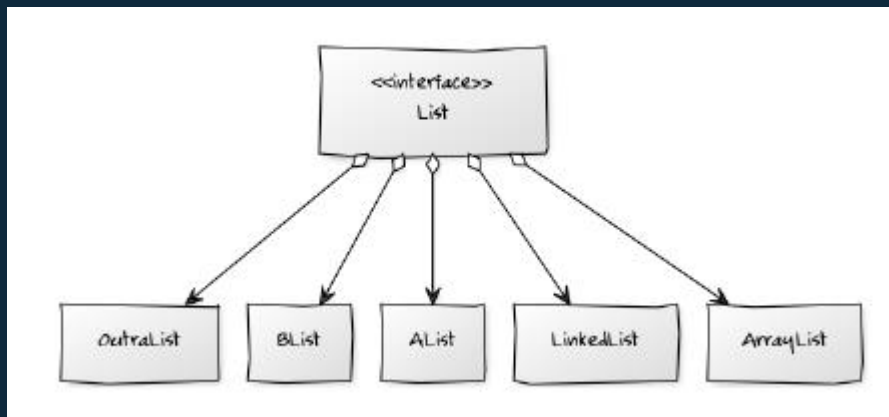
Buscando acoplamentos adequados

- ◇ Por mais estranho que pareça, é comum nos acoplarmos com classes e nem percebermos. Listas em Java ou C#, por exemplo é comum que nossos códigos acoplem-se com a interface **List**. Ou mesmo com a classe **String**, muito utilizada no dia a dia. Ao usar qualquer uma das classes, seu código passa a estar acoplado a ele.
- ◇ Mas por que acoplar-se com **List** e **String** não é problemático, mas acoplar-se com **EnviadorDeEmail** ou com qualquer outra classe que contenha uma regra de negocio é?
- ◇ Qual é a característica de **List** e qual é a característica de **String** que faz com que o acoplamento com ela seja menos dolorido do que com as outras classes? Encontrar essa característica é fundamental, pois aí bastará replicá-la; e, do mesmo jeito como não nos importamos ao acoplar com **List**, não nos importaremos em acoplar com outras classes dos nossos sistemas também.



Estabilidade de classes

- ◇ A resposta, na verdade, é que a interface **List** é **estável**.
- ◇ A **estabilidade** está relacionada com a quantidade de esforço necessário para fazer uma alteração, ou seja, classes estáveis mudam pouco e classes instáveis mudam frequentemente.
- ◇ Ela muda muito pouco, ou quase nunca muda. E, como ela quase nunca muda, ela raramente propaga mudanças para a classe principal. Esse é o tipo de “acoplamento bom”: a dependência é estável.





Estabilidade de classes

- ◇ Reforçando a ideia, se uma determinada classe depende de **List**, isso não é um problema porque ela não muda. Se ela não muda, a classe principal não sofrera impacto com a mudança dela. Este é o ponto: acoplar-se a classes, interfaces, módulos, que sejam estáveis, que tendam a mudar muito pouco.
- ◇ Interfaces são um bom caminho pra isso. Afinal, interfaces são apenas contratos: elas não tem código que pode forçar uma mudança, e geralmente tem implementações dela, e isso faz com que o desenvolvedor pense duas vezes antes de mudar o contrato.

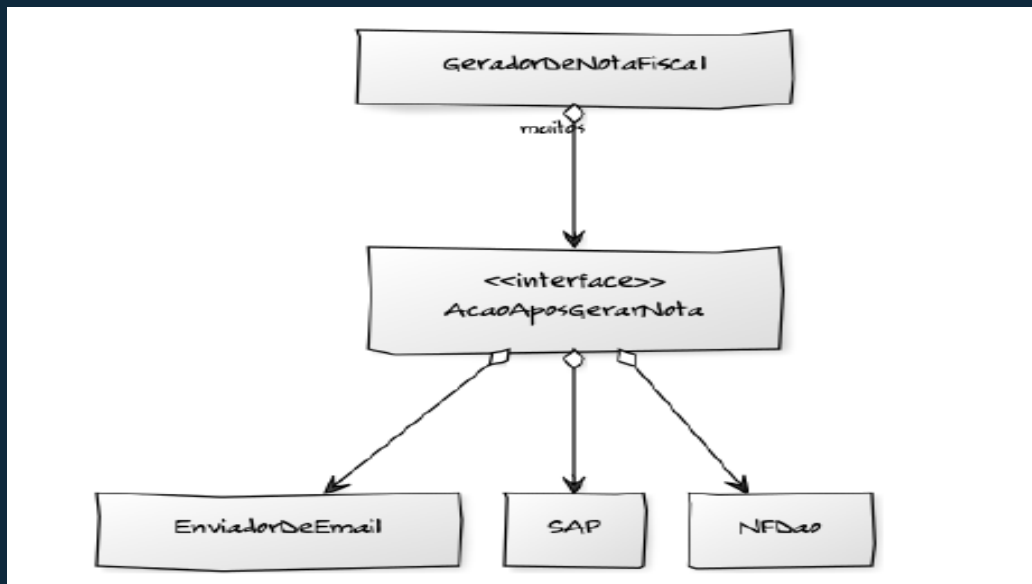




“Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações e abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações.”

Aplicando o DIP

- ◇ Veja, se criarmos uma interface **AcaoAposGerarNotaFiscal** em nosso sistema, e fizermos com que **SAP**, **EnviadorDeEmail**, **EnviadorDeSMS**, **NFDao**, ou qualquer outra ação que deva ser executada, implemente essa interface, ela será estável por natureza.




Aplicando o DIP

- ◇ Em código, a resposta para o problema seria algo como o seguinte, criaremos uma interface **AcaoAposGerarNota**, que representa a sequencia de ações que devem ser executadas apos a sua geração; a classe **GeradorDeNotaFiscal**, em vez de depender de cada ação especifica, passa a depender de uma lista de ações. Repare que o gerador agora depende apenas da interface, que, por sua vez, e bastante estável e o problema agora está controlado:

```
interface AcaoAposGerarNota {  
    void executa(NotaFiscal nf);  
}  
  
class NFDao implements AcaoAposGerarNota {  
    // implementacao  
}  
  
class QualquerOutraAcao implements AcaoAposGerarNota {  
    // implementacao  
}
```

Aplicando o DIP

```
public class GeradorDeNotaFiscal {  
  
    private final List<AcaoAposGerarNota> acoes;  
  
    public GeradorDeNotaFiscal(List<AcaoAposGeraNota> acoes) {  
        this.acoes = acoes;  
    }  
  
    public NotaFiscal gera(Fatura fatura) {  
  
        double valor = fatura.getValorMensal();  
  
        NotaFiscal nf = new NotaFiscal(  
            valor,  
            impostoSimplesSobre0(valor)  
        );  
  
        for(AcaoAposGerarNota acao : acoes) {  
            acoes.executa(nf);  
        }  
  
        return nf;  
    }  
  
    private double impostoSimplesSobre0(double valor) {  
        return valor * 0.06;  
    }  
}
```

Problemas com a violação do princípio

- ◇ Classes frágeis e fácil de quebrar.
- ◇ Propagação de erros em classes que utilizam as camadas mais internas do sistema.





Vantagens / Desvantagens


Vantagens

- ◇ Classes com menos bugs
- ◇ Melhor testabilidade
- ◇ Classes com baixo acoplamento

Desvantagens

- ◇ Mais arquivos para manipular
- ◇ Mais complexidade no código





Padrões de projeto que auxiliam na implementação do DIP

- ◇ Observer
- ◇ Visitor
- ◇ Factory Method





Conclusão do DIP

Neste capítulo, foi discutido o problema do acoplamento e a problemática propagação de mudanças que ele pode gerar. Chegamos a conclusão de que acoplar a classes estáveis, ou seja, classes que tendem a mudar pouco, é a solução para reduzir o problema do acoplamento.





Obrigado!

Alguma pergunta?

Você pode me encontrar em:

- ◇ willian_brito00@hotmail.com
- ◇ www.linkedin.com/in/willian-ferreira-brito
- ◇ github.com/Willian-Brito

