O que é SOLID: O guia completo para você entender os cinco princípios da POO

SOLID são cinco princípios da programação orientada a objetos que facilitam no desenvolvimento de softwares, tornando-os fáceis de manter e estender. Esses princípios podem ser aplicados a qualquer linguagem de POO.

O que é SOLID?

SOLID é um acrônimo criado por Michael Feathers, após observar que cinco princípios da orientação a objetos e design de código — *Criados por Robert C. Martin (Uncle Bob) e abordados no artigo* The Principles of OOD — poderiam se encaixar nesta palavra.

S.O.L.I.D: Os 5 princípios da POO

1. **S — Single Responsiblity Principle** (Princípio da responsabilidade única)
2. **O — Open-Closed Principle** (Princípio Aberto-Fechado)
3. **L — Liskov Substitution Principle** (Princípio da substituição de Liskov)
4. **I — Interface Segregation Principle** (Princípio da Segregação da Interface)
5. **D — Dependency Inversion Principle** (Princípio da inversão da dependência)

Esses princípios ajudam o programador a escrever códigos mais limpos, separando responsabilidades, diminuindo acoplamentos, facilitando na refatoração e estimulando o reaproveitamento do código.

1. SRP — Single Responsibility Principle:

Princípio da Responsabilidade Única — **Uma classe deve ter um, e somente um motivo para mudar.**

Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

Quando estamos aprendendo programação orientada a objetos, sem sabermos, damos a uma classe mais de uma responsabilidade e acabamos criando classes que fazem de tudo — *God Class**. Num primeiro momento isso pode parecer eficiente, mas como as responsabilidades acabam se

misturando, quando há necessidade de realizar alterações nessa classe, será difícil modificar uma dessas responsabilidades sem comprometer as outras. Toda alteração acaba sendo introduzida com um certo nível de incerteza em nosso sistema — principalmente se não existirem testes automatizados!

*__*God Class* — Classe Deus:__ Na programação orientada a objetos, é uma classe que sabe demais ou faz demais.

Nota: Os exemplos desse artigo foram escritos usando a linguagem **PHP**, porém, são facilmente compreendidos por qualquer pessoa que já teve contato com programação orientada a objetos, independente da linguagem.

Exemplo prático do SRP:

```php
<?php

class Order
{
    public function calculateTotalSum(){/*...*/}
    public function getItems(){/*...*/}
    public function getItemCount(){/*...*/}
    public function addItem($item){/*...*/}
    public function deleteItem($item){/*...*/}

    public function printOrder(){/*...*/}
    public function showOrder(){/*...*/}

    public function load(){/*...*/}
    public function save(){/*...*/}
    public function update(){/*...*/}
    public function delete(){/*...*/}
}

// Reference: https://www.apphp.com/tutorials/index.php?page=solid-principles-in-php-examples
```

srp-solid-violate.php hosted with ❤ by GitHub view raw

A classe *Order* viola o SRP porque realiza 3 tipos distintos de tarefas. Além de lidar com as informações do pedido, ela também é responsável pela exibição e manipulação dos dados. Lembre-se, o princípio da responsabilidade única preza que *uma classe deve ter um, e somente um, motivo para mudar***.**

A violação do Single Responsibility Principle pode gerar alguns problemas, sendo eles:

- 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;

Aplicando o SRP na classe *Order*, podemos refatorar o código da seguinte forma:

```php
<?php

class Order
{
    public function calculateTotalSum(){/*...*/}
    public function getItems(){/*...*/}
    public function getItemCount(){/*...*/}
    public function addItem($item){/*...*/}
    public function deleteItem($item){/*...*/}
}

class OrderRepository
{
    public function load($orderID){/*...*/}
    public function save($order){/*...*/}
    public function update($order){/*...*/}
    public function delete($order){/*...*/}
}

class OrderViewer
{
    public function printOrder($order){/*...*/}
    public function showOrder($order){/*...*/}
}

//Reference: https://www.apphp.com/tutorials/index.php?page=solid-principles-in-php-examples
```

srp-solid-refactor.php hosted with ❤ by GitHub view raw

Perceba no exemplo acima que agora temos 3 classes, cada uma cuidando da sua responsabilidade.

O princípio da responsabilidade única não se limita somente a classes, ele também pode ser aplicado em métodos e funções, ou seja, tudo que é responsável por executar uma ação, deve ser responsável por apenas aquilo que se propõe a fazer.

Considero o SRP um dos princípios mais importantes, ele acaba sendo a base para outros princípios e padrões porque aborda temas como acoplamento e coesão, características que todo código orientado a objetos deveria ter.

Aplicando esse princípio, automaticamente você estará escrevendo um código mais limpo e de fácil manutenção! Se você tem interesse nesse assunto, recomendo a leitura das **boas práticas para escrever códigos impecáveis.**

2. OCP — Open-Closed Principle:

Princípio Aberto-Fechado — **Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação**, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.

Exemplo prático do OCP:

Em um sistema hipotético de RH, temos duas classes que representam os contratos de trabalhos dos funcionários de uma pequena empresa, contratados e estagiários. Além de uma classe para processar a folha de pagamento.

```php
<?php

class ContratoClt
{
    public function salario()
    {
        //...
    }
}

class Estagio
{
    public function bolsaAuxilio()
    {
        //...
    }
}

class FolhaDePagamento
{
    protected $saldo;

    public function calcular($funcionario)
    {
        if ( $funcionario instanceof ContratoClt ) {
            $this->saldo = $funcionario->salario();
        } else if ( $funcionario instanceof Estagio) {
            $this->saldo = $funcionario->bolsaAuxilio();
        }
    }
}
```

ocp-solid-violate.php hosted with ❤ by GitHub view raw

A classe `FolhaDePagamento` precisa verificar o funcionário para aplicar a regra de negócio correta na hora do pagamento. Supondo que a empresa cresceu e resolveu trabalhar com funcionários PJ, obviamente seria necessário modificar essa classe! Sendo assim, estaríamos quebrando o princípio Open-Closed do SOLID.

Qual o problema de se alterar a classe *FolhaDePagamento?*

Não seria mais fácil apenas acrescentar mais um *IF* e verificar o novo tipo de funcionário PJ aplicando as respectivas regras? Sim, e provavelmente essa seria a solução que programadores menos experientes iriam fazer. Mas, esse é exatamente o problema! **Alterar uma classe já existente para adicionar um novo comportamento, corremos um sério risco de introduzir bugs em algo que já estava funcionando.**

Lembre-se: OCP *preza que uma classe deve estar fechada para alteração e aberta para extensão.*

Como adicionamos um novo comportamento sem alterar o código fonte já existente?

O guru Uncle Bob resumiu a solução em uma frase:

“Separate extensible behavior behind an interface, and flip the dependencies.”

Em tradução direta, seria:

“Separe o comportamento extensível por trás de uma interface e inverta as dependências.”

O que devemos fazer é concentrar nos aspectos essências do contexto, abstraindo-os para uma interface. Se as abstrações são bem definidas, logo o software estará aberto para extensão.

Aplicando OCP na prática

Voltando para o nosso exemplo, podemos concluir que o contexto que estamos lidando é a remuneração dos contratos de trabalho, aplicando as premissas de se *isolar o comportamento extensível atrás de uma interface*, podemos criar uma interface com o nome `Remuneravel` contendo o método `remuneracao()`, e fazer com que nossas classes de contrato de trabalho implementem essa interface. Além disso, iremos colocar as regras de calculo de remuneração para suas respectivas classes, dentro do método `remuneracao()`, fazendo com que a classe `FolhaDePagamento` dependa somente da interface `Remuneravel` que iremos criar.

Veja o código refatorado abaixo:

```php
<?php

interface Remuneravel
{
    public function remuneracao();
}

class ContratoClt implements Remuneravel
{
    public function remuneracao()
    {
        //...
    }
}

class Estagio implements Remuneravel
{
    public function remuneracao()
    {
        //...
    }
}

class FolhaDePagamento
{
    protected $saldo;

    public function calcular(Remuneravel $funcionario)
    {
        $this->saldo = $funcionario->remuneracao();
    }
}
```

ocp-solid-example.php hosted with ❤ by GitHub view raw

Agora a classe `FolhaDePagamento` não precisa mais saber quais métodos chamar para calcular. Ela será capaz de calcular o pagamento corretamente de qualquer novo tipo de funcionário que seja criado no futuro (*ContratoPJ*) — *desde que ele implemente a interface* `Remuneravel` — sem qualquer necessidade de alteração do seu código fonte. Dessa forma, acabamos de implementar o princípio de Aberto-Fechado do SOLID em nosso código!

Open-Closed Principle também é base para o padrão de projeto Strategy — a sua principal vantagem é a facilidade na adição de novos requisitos, diminuindo as chances de introduzir novos bugs — *ou bugs de menor expressão* — pois o novo comportamento fica isolado, e o que estava funcionando provavelmente continuara funcionando.

3. LSP— Liskov Substitution Principle:

Princípio da substituição de Liskov — **Uma classe derivada deve ser substituível por sua classe base**.

O princípio da substituição de Liskov foi introduzido por Barbara Liskov em sua conferência “Data abstraction” em 1987. 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

Um exemplo mais simples e de fácil compreensão dessa definição. Seria:

se S é um subtipo de T, então os objetos do tipo T, em um programa, podem ser substituídos pelos objetos de tipo S

sem que seja necessário alterar as propriedades deste programa.

Para facilitar o entendimento, veja esse princípio na prática neste simples exemplo:

```php
<?php

class A
{
    public function getNome()
    {
        echo 'Meu nome é A';
    }
}

class B extends A
{
    public function getNome()
    {
        echo 'Meu nome é B';
    }
}

$objeto1 = new A;
$objeto2 = new B;

function imprimeNome(A $objeto)
{
    return $objeto->getNome();
}

imprimeNome($objeto1); // Meu nome é A
imprimeNome($objeto2); // Meu nome é B
```

lsp-solid-example.php hosted with ❤ by GitHub view raw

Estamos passando como parâmetro tanto a classe pai como a classe derivada e o código continua funcionando da forma esperada.

Exemplos de violação do LSP:

- Sobrescrever/implementar um método que não faz nada;

- Lançar uma exceção inesperada;
- Retornar valores de tipos diferentes da classe base;

```php
<?php

# - Sobrescrevendo um método que não faz nada...
class Voluntario extends ContratoDeTrabalho
{
    public function remuneracao()
    {
        // não faz nada
    }
}


# - Lançando uma exceção inesperada...
class MusicPlay
{
    public function play($file)
    {
        // toca a música
    }
}

class Mp3MusicPlay extends MusicPlay
{
    public function play($file)
    {
        if (pathinfo($file, PATHINFO_EXTENSION) !== 'mp3') {
            throw new Exception;
        }

        // toca a música
    }
}


# - Retornando valores de tipos diferentes...
class Auth
{
    public function checkCredentials($login, $password)
    {
        // faz alguma coisa

        return true;
    }
}

class AuthApi extends Auth
{
    public function checkCredentials($login, $password)
    {
        // faz alguma coisa

        return ['auth' => true, 'status' => 200];
    }
}
```

Para não violar o Liskov Substitution Principle, além de estruturar muito bem as suas abstrações, em alguns casos, você precisara usar a *injeção de dependência* e também usar outros princípios do SOLID, como por exemplo, o *Open-Closed Principle* e o *Interface Segregation Principle* — será abordado no próximo tópico.

Seguir o **LSP** nos permite usar o polimorfismo com mais confiança. Podemos chamar nossas classes derivadas referindo-se à sua classe base sem preocupações com resultados inesperados.

4. ISP — Interface Segregation Principle:

Princípio da Segregação da Interface — **Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.**

Esse princípio basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica.

Vamos ver o ISP na prática através de códigos:

Em um cenário fictício para criação de um game de animais, teremos algumas aves que serão tratadas como personagens dentro do jogo. Sendo assim, criaremos uma interface `Aves` para abstrair o comportamento desses animais, depois faremos que nossas classes implementem essa interface, veja:

```php
<?php

interface Aves
{
    public function setLocalizacao($longitude, $latitude);
    public function setAltitude($altitude);
    public function renderizar();
}

class Papagaio implements Aves
{
    public function setLocalizacao($longitude, $latitude)
    {
        //Faz alguma coisa
    }

    public function setAltitude($altitude)
    {
        //Faz alguma coisa
    }

    public function renderizar()
    {
        //Faz alguma coisa
    }
}

class Pinguim implements Aves
{
    public function setLocalizacao($longitude, $latitude)
    {
        //Faz alguma coisa
    }

    // A Interface Aves está forçando a Classe Pinguim a implementar esse método.
    // Isso viola o príncipio ISP
    public function setAltitude($altitude)
    {
        //Não faz nada...  Pinguins são aves que não voam!
    }

    public function renderizar()
    {
        //Faz alguma coisa
    }
}
```

isp-solid-violate.php hosted with ❤ by GitHub view raw

Percebam que ao criar a interface `Aves`, atribuímos comportamentos genéricos e isso acabou forçando a classe `Pinguim` a implementar o método `setAltitude()` do qual ela não deveria ter, pois pinguins não voam! Dessa forma, estamos violando o *Interface Segregation Principle* — E o **LSP** também!

Para resolver esse problema, devemos criar interfaces mais específicas, veja:

```php
<?php

interface Aves
{
    public function setLocalizacao($longitude, $latitude);
    public function renderizar();
}

interface AvesQueVoam extends Aves
{
    public function setAltitude($altitude);
}

class Papagaio implements AvesQueVoam
{
    public function setLocalizacao($longitude, $latitude)
    {
        //Faz alguma coisa
    }

    public function setAltitude($altitude)
    {
        //Faz alguma coisa
    }

    public function renderizar()
    {
        //Faz alguma coisa
    }
}

class Pinguim implements Aves
{
    public function setLocalizacao($longitude, $latitude)
    {
        //Faz alguma coisa
    }

    public function renderizar()
    {
        //Faz alguma coisa
    }
}
```

isp-solid-example.php hosted with ❤ by GitHub view raw

No exemplo acima, retiramos o método `setAltitude()` da interface `Aves` e adicionamos em uma nova interface derivada `AvesQueVoam`. Isso nos permitiu isolar os comportamentos das aves de maneira correta dentro do jogo, respeitando o princípio da segregação das interfaces.

5. DIP — Dependency Inversion Principle:

Princípio da Inversão de Dependência — **Dependa de abstrações e não de implementações.**

De acordo com Uncle Bob, esse princípio pode ser definido da seguinte forma:

1. “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.”

2. “Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.”

No contexto da programação orientada a objetivos, é comum que as pessoas confundam a *Inversão de Dependência* com a *Injeção de Dependência*, porém são coisas distintas, mas que relacionam entre si com um proposito em comum, deixar o código desacoplado.

Importante: *Inversão de Dependência* **não é igual** a *Injeção de Dependência,* **fique ciente disso**! A I*nversão de Dependência* é um princípio (Conceito) e a I*njeção de Dependência* é um padrão de projeto (D*esign Pattern*).

Vamos entender tudo isso na prática através de exemplos:

```php
<?php

use MySQLConnection;

class PasswordReminder
{
    private $dbConnection;

    public function __construct()
    {
        $this->dbConnection = new MySQLConnection();
    }

    // Faz alguma coisa
}
```

acouple-code.php hosted with ❤ by GitHub view raw

Para recuperar a senha, a classe `PasswordReminder`, precisa conectar na base de dados, por tanto, ela cria um instancia da classe `MySQLConnection` em seu método construtor para realizar as respectivas operações.

Nesse pequeno trecho de código temos um **alto nível de acoplamento,** isso acontece porque a classe `PasswordReminder` tem a responsabilidade de criar uma instância da classe `MySQLConnection`*!* *S*e quiséssemos reaproveitar essa classe em outro sistema**,** teriamos obrigatoriamente de levar a classe `MySQLConnection` junto, portanto, temos um forte acoplamento aqui.

Para resolver esse problema de acoplamento, podemos refatorar nosso código da seguinte forma. Veja:

```php
<?php

use MySQLConnection;

class PasswordReminder
{
    private $dbConnection;

    public function __construct(MySQLConnection $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }

    // Faz alguma coisa
}
```

dependency-injection-example.php hosted with ❤ by GitHub view raw

Com o código refatorado, a criação do objeto `MySQLConnection` deixa de ser uma responsabilidade da classe `PasswordReminder`, a classe de conexão com o banco de dados virou uma dependência que deve ser injetada via método construtor. Olha o que apareceu para nós: **Injeção de Dependência!**

Apesar de termos usado a injeção de dependência para melhorar o nível de acoplamento do nosso código, ele ainda viola o princípio da inversão de dependência! — lembre-se, um não é igual ao outro.

Além de violar o **DIP**, se você prestar atenção na forma que o exemplo foi codificado irá perceber que ele também viola o **Open-Closed Principle.** Por exemplo, se precisarmos alterar o banco de dados de *MySQL* para *Oracle* teríamos que editar a classe `PasswordReminder`.

Por que nosso exemplo refatorado viola o Dependency Inversion Principle?

Porque estamos dependendo de uma implementação e não de uma abstração, simples assim.

De acordo com a definição do **DIP**, ***um módulo de alto nível não deve depender de módulos de baixo nível, ambos devem depender da abstração***. Então, a primeira coisa que precisamos fazer é identificar no nosso código qual é o módulo de alto nível e qual é o módulo de baixo nível. Módulo de alto nível é um módulo que depende de outros módulos.

No nosso exemplo, `PasswordReminder` depende da classe `MySQLConnection`. Sendo assim, `PasswordReminder` é o módulo de alto nível e `MySQLConnection` é o módulo de baixo nível.

Mas, `MySQLConnection` é uma implementação e não uma abstração!

Como refatoramos nosso exemplo para utilizar o DIP?

Se tratando de POO, você já ouviu aquela frase:

"Programe para uma interface e não para uma implementação."

Pois bem, é exatamente o que iremos fazer, criar uma interface!

```
interface DBConnectionInterface
{
    public function connect();
}
```

Agora, vamos refatorar nosso exemplo fazendo que nossos módulos de alto e baixo nível dependam da abstração proposta pela interface que acabamos de criar. Veja:

```php
<?php

interface DBConnectionInterface
{
    public function connect();
}


class MySQLConnection implements DBConnectionInterface
{
    public function connect()
    {
        // ...
    }
}

class OracleConnection implements DBConnectionInterface
{
    public function connect()
    {
        // ...
    }
}

class PasswordReminder
{
    private $dbConnection;

    public function __construct(DBConnectionInterface $dbConnection) {
        $this->dbConnection = $dbConnection;
    }

    // Faz alguma coisa
}
```

dip-solid-example.php hosted with ❤ by GitHub view raw

Perfeito! Agora a classe `PasswordReminder` não tem a mínima ideia de qual banco de dados a aplicação irá utilizar. Dessa forma, não estamos mais violando o **DIP**, ambas as classes estão desacopladas e dependendo de uma abstração. Além disso, estamos favorecendo a reusabilidade do código e como *"bônus"* também estamos respeitando o **SRP** e o **OCP**.

Conclusão

A sistemática dos princípios **SOLID** tornam o software mais robusto, escalável e flexível, deixando-o tolerante a mudanças, facilitando a implementação de novos requisitos para a evolução e manutenção do sistema.