



Published in Desenvolvendo com Paixão



João Roberto da Paixão

Follow

Jan 6, 2019 · 10 min read



Save



O que é SOLID: O guia completo para você entender os 5 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.

SOLID

Princípios da programação orientada a objetos

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

(a.k.a. *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 Responsibility 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:

```
1  <?php
2
3  class Order
4  {
5      public function calculateTotalSum(){/*...*/}
6      public function getItems(){/*...*/}
7      public function getItemCount(){/*...*/}
8      public function addItem($item){/*...*/}
9      public function deleteItem($item){/*...*/}
10
11     public function printOrder(){/*...*/}
12     public function showOrder(){/*...*/}
13
14     public function load(){/*...*/}
15     public function save(){/*...*/}
16     public function update(){/*...*/}
17     public function delete(){/*...*/}
18 }
19
20 // Reference: https://www.apphp.com/tutorials/index.php?page=solid-principles-in-php-ex
```

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:

```
1  <?php
2
3  class Order
4  {
5      public function calculateTotalSum(){/*...*/}
6      public function getItems(){/*...*/}
7      public function getItemCount(){/*...*/}
8      public function addItem($item){/*...*/}
9      public function deleteItem($item){/*...*/}
10 }
11
12 class OrderRepository
13 {
14     public function load($orderId){/*...*/}
15     public function save($order){/*...*/}
16     public function update($order){/*...*/}
17     public function delete($order){/*...*/}
18 }
19
20 class OrderViewer
21 {
22     public function printOrder($order){/*...*/}
23     public function showOrder($order){/*...*/}
24 }
25
26 //Reference: https://www.appph.com/tutorials/index.php?page=solid-principles-in-php-exa
```

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.

```
1  <?php
2
3  //Ruim:
4  function emailClients(array $clients): void
5  {
6      foreach ($clients as $client) {
7          $clientRecord = $db->find($client);
8          if ($clientRecord->isActive()) {
9              email($client);
10         }
11     }
12 }
13
14
15
16 // Bom:
17 function emailClients(array $clients): void
18 {
19     $activeClients = activeClients($clients);
20     array_walk($activeClients, 'email');
21 }
22
23 function activeClients(array $clients): array
24 {
25     return array_filter($clients, 'isClientActive');
26 }
27
28 function isClientActive(int $client): bool
29 {
30     $clientRecord = $db->find($client);
31
32     return $clientRecord->isActive();
33 }
34
35
36 // Reference: https://github.com/jupeter/clean-code-php
```

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.


```
1  <?php
2
3  class ContratoClt
4  {
5      public function salario()
6      {
7          //...
8      }
9  }
10
11 class Estagio
12 {
13     public function bolsaAuxilio()
14     {
15         //...
16     }
17 }
18
19 class FolhaDePagamento
20 {
21     protected $saldo;
22
23     public function calcular($funcionario)
24     {
25         if ( $funcionario instanceof ContratoClt ) {
26             $this->saldo = $funcionario->salario();
27         } else if ( $funcionario instanceof Estagio ) {
28             $this->saldo = $funcionario->bolsaAuxilio();
29         }
30     }
31 }
```

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 essenciais 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:

```
1  <?php
2
3  interface Remuneravel
4  {
5      public function remuneracao();
6  }
7
8  class ContratoClt implements Remuneravel
9  {
10     public function remuneracao()
11     {
12         //...
13     }
14 }
15
16 class Estagio implements Remuneravel
17 {
18     public function remuneracao()
19     {
20         //...
21     }
22 }
23
24 class FolhaDePagamento
25 {
26     protected $saldo;
27
28     public function calcular(Remuneravel $funcionario)
29     {
30         $this->saldo = $funcionario->remuneracao();
31     }
32 }
```

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. — *Falarei desse padrão em um próximo artigo*. Particularmente esse é o **princípio que eu mais admiro**, 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. — Wikipedia.

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

```
1  <?php
2
3  class A
4  {
5      public function getNome()
6      {
7          echo 'Meu nome é A';
8      }
9  }
10
11 class B extends A
12 {
13     public function getNome()
14     {
15         echo 'Meu nome é B';
16     }
17 }
18
19 $objeto1 = new A;
20 $objeto2 = new B;
21
22 function imprimeNome(A $objeto)
23 {
24     return $objeto->getNome();
25 }
26
27 imprimeNome($objeto1); // Meu nome é A
28 imprimeNome($objeto2); // Meu nome é B
```

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;

```
1  <?php
2
3  # - Sobrescrevendo um método que não faz nada...
4  class Voluntario extends ContratoDeTrabalho
5  {
6      public function remuneracao()
7      {
8          // não faz nada
9      }
10 }
11
12
13 # - Lançando uma exceção inesperada...
14 class MusicPlay
15 {
16     public function play($file)
17     {
18         // toca a música
19     }
20 }
21
22 class Mp3MusicPlay extends MusicPlay
23 {
24     public function play($file)
25     {
26         if (pathinfo($file, PATHINFO_EXTENSION) !== 'mp3') {
27             throw new Exception;
28         }
29
30         // toca a música
31     }
32 }
33
34
35 # - Retornando valores de tipos diferentes...
36 class Auth
37 {
38     public function checkCredentials($login, $password)
39     {
40         // faz alguma coisa
41
42         return true;
43     }
44 }
45
46 class AuthApi extends Auth
47 {
48     public function checkCredentials($login, $password)
```



```
48     public function checkCredentials($login, $password)
49     {
50         // faz alguma coisa
51
52         return ['auth' => true, 'status' => 200];
53     }
54 }
```

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

[view raw](#)

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:

```
1  <?php
2
3  interface Aves
4  {
5      public function setLocalizacao($longitude, $latitude);
6      public function setAltitude($altitude);
7      public function renderizar();
8  }
9
10 class Papagaio implements Aves
11 {
12     public function setLocalizacao($longitude, $latitude)
13     {
14         //Faz alguma coisa
15     }
16
17     public function setAltitude($altitude)
18     {
19         //Faz alguma coisa
20     }
21
22     public function renderizar()
23     {
24         //Faz alguma coisa
25     }
26 }
27
28 class Pinguim implements Aves
29 {
30     public function setLocalizacao($longitude, $latitude)
31     {
32         //Faz alguma coisa
33     }
34
35     // A Interface Aves está forçando a Classe Pinguim a implementar esse método.
36     // Isso viola o princípio ISP
37     public function setAltitude($altitude)
38     {
39         //Não faz nada... Pinguins são aves que não voam!
40     }
41
42     public function renderizar()
43     {
44         //Faz alguma coisa
45     }
46 }
```

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:

```
1  <?php
2
3  interface Aves
4  {
5      public function setLocalizacao($longitude, $latitude);
6      public function renderizar();
7  }
8
9  interface AvesQueVoam extends Aves
10 {
11     public function setAltitude($altitude);
12 }
13
14 class Papagaio implements AvesQueVoam
15 {
16     public function setLocalizacao($longitude, $latitude)
17     {
18         //Faz alguma coisa
19     }
20
21     public function setAltitude($altitude)
22     {
23         //Faz alguma coisa
24     }
25
26     public function renderizar()
27     {
28         //Faz alguma coisa
29     }
30 }
31
32 class Pinguim implements Aves
33 {
34     public function setLocalizacao($longitude, $latitude)
35     {
36         //Faz alguma coisa
37     }
38
39     public function renderizar()
40     {
41         //Faz alguma coisa
42     }
43 }
```

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

Poderíamos melhorar ainda mais esse exemplo, criando uma interface `Renderizavel` pra abstrair esse comportamento, mas o foco aqui é explicar o conceito e não desenvolver o game, então vamos para o próximo princípio.

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 objetos, 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 propósito em comum, deixar o código desacoplado.

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

Vamos entender tudo isso na prática através de exemplos:

```

1  <?php
2
3  use MySqlConnection;
4
5  class PasswordReminder
6  {
7      private $dbConnection;
8
9      public function __construct()
10     {
11         $this->dbConnection = new MySqlConnection();
12     }
13
14     // Faz alguma coisa
15 }

```

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`. Se quiséssemos reorganizar essa classe em outro sistema

Open in app ↗

[Sign up](#)

[Sign In](#)



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

```

1  <?php
2
3  use MySqlConnection;
4
5  class PasswordReminder
6  {
7      private $dbConnection;
8
9      public function __construct(MySqlConnection $dbConnection)
10     {
11         $this->dbConnection = $dbConnection;
12     }
13
14     // Faz alguma coisa
15 }

```

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**



6.7K



41

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:

“Programa 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:


```
1  <?php
2
3  interface DBConnectionInterface
4  {
5      public function connect();
6  }
7
8
9  class MySQLConnection implements DBConnectionInterface
10 {
11     public function connect()
12     {
13         // ...
14     }
15 }
16
17 class OracleConnection implements DBConnectionInterface
18 {
19     public function connect()
20     {
21         // ...
22     }
23 }
24
25 class PasswordReminder
26 {
27     private $dbConnection;
28
29     public function __construct(DBConnectionInterface $dbConnection) {
30         $this->dbConnection = $dbConnection;
31     }
32
33     // Faz alguma coisa
34 }
```

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.

Levando em consideração algumas experiências vividas ao longo da **minha história no mundo da tecnologia**, acredito que os princípios **SOLID**, juntamente com algumas técnicas e **boas praticas de Clean Code**, são fatores essenciais que todos os desenvolvedores deveriam aplicar em seus códigos.

Pode ser um pouco assustador no início usar todos esses princípios — *nem sempre conseguiremos aplicar todos em nosso projeto* — mas com a prática e constância, aos poucos vamos adquirindo a experiência necessária para escrever códigos cada vez mais maduros, os princípios **SOLID** servem como guias pra isso.

Obrigado por ler! Se você gostou do artigo, dê um clap 🙌.

Referências:

- <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- <https://en.wikipedia.org/wiki/SOLID>
- <https://laracasts.com/series/solid-principles-in-php>
- <https://www.tomdalling.com/blog/software-design/solid-class-design-the-liskov-substitution-principle>

Artigos que talvez você possa se interessar:

#1 — Clean Code: O que é? Porque usar?	
--	--