habr.com

Шпаргалка по SOLID-принципам с примерами на PHP

15-21 минута

Тема SOLID-принципов и в целом чистоты кода не раз поднималась на Хабре и, возможно, уже порядком изъезженная. Но тем не менее, не так давно мне приходилось проходить собеседования в одну интересную IT-компанию, где меня попросили рассказать о принципах SOLID с примерами и ситуациями, когда я не соблюл эти принципы и к чему это привело. И в тот момент я понял, что на каком-то подсознательном уровне я понимаю эти принципы и даже могут назвать их все, но привести лаконичные и понятные примеры для меня стало проблемой. Поэтому я и решил для себя самого и для сообщества обобщить информацию по SOLID-принципам для ещё лучшего её понимания. Статья должна быть полезной, для людей только знакомящихся с SOLID-принципами, также, как и для людей «съевших собаку» на SOLID-принципах.

Для тех, кто знаком с принципами и хочет только освежить память о них и их использовании, можно обратиться сразу к шпаргалке в конце статьи.

Что же такое SOLID-принципы? Если верить определению Wikipedia, это:

> аббревиатура пяти основных принципов дизайна классов в объектно-ориентированном проектировании — **S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface

segregation и **D**ependency inversion.

Таким образом, мы имеем 5 принципов, которые и рассмотрим ниже:

- Принцип единственной ответственности (Single responsibility)
- Принцип открытости/закрытости (Open-closed)
- Принцип подстановки Барбары Лисков (Liskov substitution)
- Принцип разделения интерфейса (Interface segregation)
- Принцип инверсии зависимостей (Dependency Invertion)

Принцип единственной ответственности (Single responsibility)

Итак, в качества примера возьмём довольно популярный и широкоиспользуемый пример — интернет-магазин с заказами, товарами и покупателями.

Принцип единственной ответственности гласит — *«На каждый объект должна быть возложена одна единственная обязанность»*. Т.е. другими словами — конкретный класс должен решать конкретную задачу — ни больше, ни меньше.

Рассмотрим следующее описание класса для представления заказа в интернет-магазине:

```
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(){}
}
```

Как можно увидеть, данный класс выполняет операций для 3 различный типов задач: работа с самим заказом(`calculateTotalSum, getItems, getItemsCount, addItem, deleteItem`), отображение заказа(`printOrder, showOrder`) и работа с хранилищем данных(`load, save, update, delete`).

К чему это может привести?

Приводит это к тому, что в случае, если мы хотим внести изменения в методы печати или работы хранилища, мы изменяем сам класс заказа, что может привести к его неработоспособности.

Решить эту проблему стоит разделением данного класса на 3 отдельных класса, каждый из которых будет заниматься своей задачей

```
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){}
}
```

Теперь каждый класс занимается своей конкретной задачей и для каждого класса есть только 1 причина для его изменения.

Принцип открытости/закрытости (Open-closed)

Данный принцип гласит — "программные сущности должны быть открыты для расширения, но закрыты для модификации". На более простых словах это можно описать так — все классы, функции и т.д. должны проектироваться так, чтобы для изменения их поведения, нам не нужно было изменять их исходный код.
Рассмотри на примере класса `OrderRepository`.

```
class OrderRepository
{
        public function load($orderID)
        {
                $pdo = new
PDO($this->config->getDsn(),
$this->config->getDBUser(),
$this->config->getDBPassword());
                $statement = $pdo->prepare('SELECT
* FROM `orders` WHERE id=:id');
                $statement->execute(array(':id' =>
$orderID));
```

```
                return
$query->fetchObject('Order');
        }
        public function save($order){}
        public function update($order){}
        public function delete($order){}
}
```

В данном случае хранилищем у нас является база данных. например, MySQL. Но вдруг мы захотели подгружать наши данные о заказах, например, через API стороннего сервера, который, допустим, берёт данные из 1С. Какие изменения нам надо будет внести? Есть несколько вариантов, например, непосредственно изменить методы класса `OrderRepository`, но этот не соответствует *принципу открытости/закрытости*, так как класс закрыт для модификации, да и внесение изменений в уже хорошо работающий класс нежелательно. Значит, можно наследоваться от класса `OrderRepository` и переопределить все методы, но это решение не самое лучше, так как при добавлении метода в `OrderRepository` нам придётся добавить аналогичные методы во все его наследники. Поэтому для выполнения *принципа открытости/закрытости* лучше применить следующее решение — создать интерфейс `IOrderSource`, который будет реализовываться соответствующими классами `MySQLOrderSource`, `ApiOrderSource` и так далее.

Интерфейс IOrderSource и его реализация и использование

```
class OrderRepository
{
        private $source;

        public function setSource(IOrderSource
$source)
```

```
        {
                $this->source = $source;
        }

        public function load($orderID)
        {
                return
$this->source->load($orderID);
        }
        public function save($order){}
        public function update($order){}
}

interface IOrderSource
{
        public function load($orderID);
        public function save($order);
        public function update($order);
        public function delete($order);
}

class MySQLOrderSource implements IOrderSource
{
        public function load($orderID);
        public function save($order){}
        public function update($order){}
        public function delete($order){}
}

class ApiOrderSource implements IOrderSource
{
        public function load($orderID);
        public function save($order){}
        public function update($order){}
```

```
        public function delete($order){}
}
```

Таким образом, мы можем изменить источник и соответственно поведение для класса `OrderRepository`, установив нужный нам класс реализующий `IOrderSource`, без изменения класса `OrderRepository`.

Принцип подстановки Барбары Лисков (Liskov substitution)

Пожалуй, принцип, который вызывает самые большие затруднения в понимании.
Принцип гласит — *«Объекты в программе могут быть заменены их наследниками без изменения свойств программы»*. Своими словами я бы это сказал так — при использовании наследника класса результат выполнения кода должен быть предсказуем и не изменять свойств метод.
К сожалению, придумать доступного примера для это принципа в рамках задачи интернет-магазина я не смог, но есть классический пример с иерархией геометрических фигур и вычисления площади. Код примера ниже.

Пример иерархии прямоугольника и квадрата и вычислении их площади

```
class Rectangle
{
        protected $width;
        protected $height;

        public setWidth($width)
        {
                $this->width = $width;
        }
```

```
        public setHeight($height)
        {
                $this->height = $height;
        }

        public function getWidth()
        {
                return $this->width;
        }

        public function getHeight()
        {
                return $this->height;
        }
}

class Square extends Rectangle
{
        public setWidth($width)
        {
                parent::setWidth($width);
                parent::setHeight($width);
        }

        public setHeight($height)
        {
                parent::setHeight($height);
                parent::setWidth($height);
        }
}

function calculateRectangleSquare(Rectangle
$rectangle, $width, $height)
{
```

```
        $rectangle->setWidth($width);
        $rectangle->setHeight($height);
        return $rectangle->getHeight *
$rectangle->getWidth;
}

calculateRectangleSquare(new Rectangle, 4, 5);
calculateRectangleSquare(new Square, 4, 5);
```

Очевидно, что такой код явно выполняется не так, как от него этого ждут.
Но в чём проблема? Разве «квадрат» не является «прямоугольником»? Является, но в геометрических понятиях. В понятиях же объектов, квадрат не есть прямоугольник, поскольку поведение объекта «квадрат» не согласуется с поведением объекта «прямоугольник».

Тогда же как решить проблему?
Решение тесно связано с таким понятием как *проектирование по контракту*. Описание проектирования по контракту может занять не одну статью, поэтому ограничимся особенностями, которые касаются *принципа Лисков*.
Проектирование по контракту ведет к некоторым ограничениям на то, как контракты могут взаимодействовать с наследованием, а именно:

- Предусловия не могут быть усилены в подклассе.
- Постусловия не могут быть ослаблены в подклассе.

«Что ещё за пред- и постусловия?» — можете спросите Вы.
Ответ: *предусловия* – это то, что должно быть выполнено вызывающей стороной перед вызовом метода, *постусловия* – это то, что, гарантируется вызываемым методом.

Вернёмся к нашему примеру и посмотрим, как мы изменили пред- и постусловия.

Предусловия мы никак не использовали при вызове методов установки высоты и ширины, а вот постусловия в классе-наследнике мы изменили и изменили на более слабые, чего по принципу Лисков делать было нельзя.
Ослабили мы их вот почему. Если за постусловие метода `setWidth` принять `(($this->width == $width) && ($this->height == $oldHeight))` (`$oldHeight` мы присвоили вначале метода setWidth), то это условие не выполняется в дочернем классе и соответственно мы его ослабили и `принцип Лисков` нарушен.

Поэтому, лучше в рамках ООП и задачи расчёта площади фигуры не делать иерархию «квадрат» наследует «прямоугольник», а сделать их как 2 отдельные сущности:

```
class Rectangle
{
        protected $width;
        protected $height;

        public setWidth($width)
        {
                $this->width = $width;
        }

        public setHeight($height)
        {
                $this->height = $height;
        }

        public function getWidth()
        {
                return $this->width;
        }
```

```
        public function getHeight()
        {
                return $this->height;
        }
}

class Square
{
        protected $size;

        public setSize($size)
        {
                $this->size = $size;
        }

        public function getSize()
        {
                return $this->size;
        }
}
```

Хороший **реальный** пример несоблюдения принципа Лискоу и решения, принятого в связи с этим, рассмотрен в книге Роберта Мартина «Быстрая разработка программ» в разделе «Принцип подстановки Лискоу. Реальный пример».

Принцип разделения интерфейса (Interface segregation)

Данный принцип гласит, что *«Много специализированных интерфейсов лучше, чем один универсальный»*
Соблюдение этого принципа необходимо для того, чтобы классы-клиенты использующий/реализующий интерфейс знали только о тех методах, которые они используют, что ведёт к уменьшению количества неиспользуемого кода.

Вернёмся к примеру с интернет-магазином.
Предположим наши товары могут иметь промокод, скидку, у них есть какая-то цена, состояние и т.д. Если это одежда то для неё устанавливается из какого материала сделана, цвет и размер.
Опишем следующий интерфейс

```
interface IItem
{
        public function applyDiscount($discount);
        public function
applyPromocode($promocode);

        public function setColor($color);
        public function setSize($size);

        public function setCondition($condition);
        public function setPrice($price);
}
```

Данный интефейс плох тем, что он включает слишком много методов. А что, если наш класс товаров не может иметь скидок или промокодов, либо для него нет смысла устанавливать материал из которого сделан (например, для книг). Таким образом, чтобы не реализовывать в каждом классе неиспользуемые в нём методы, лучше разбить интерфейс на несколько мелких и каждым конкретным классом реализовывать нужные интерфейсы.

Разбиение интерфейса IItem на несколько

```
interface IItem
{
        public function setCondition($condition);
        public function setPrice($price);
}

interface IClothes
```

```
{
        public function setColor($color);
        public function setSize($size);
        public function setMaterial($material);
}

interface IDiscountable
{
        public function applyDiscount($discount);
        public function
applyPromocode($promocode);
}

class Book implemets IItem, IDiscountable
{
    public function setCondition($condition){}
    public function setPrice($price){}
    public function applyDiscount($discount){}
    public function applyPromocode($promocode){}
}

class KidsClothes implemets IItem, IClothes
{
    public function setCondition($condition){}
    public function setPrice($price){}
    public function setColor($color){}
    public function setSize($size){}
    public function setMaterial($material){}
}
```

Принцип инверсии зависимостей (Dependency Invertion)

Принцип гласит — *«Зависимости внутри системы строятся*

на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций». Данное определение можно сократить — *«зависимости должны строится относительно абстракций, а не деталей»*.

Для примера рассмотрим оплату заказа покупателем.

```
class Customer
{
        private $currentOrder = null;

        public function buyItems()
        {
                if(is_null($this->currentOrder)){
                        return false;
                }

                $processor = new OrderProcessor();
                return
$processor->checkout($this->currentOrder);
        }

        public function addItem($item){
                if(is_null($this->currentOrder)){
                        $this->currentOrder = new
Order();
                }
                return
$this->currentOrder->addItem($item);
        }
        public function deleteItem($item){
                if(is_null($this->currentOrder)){
                        return false;
                }
```

```
                return $this->currentOrder
->deleteItem($item);
        }
}

class OrderProcessor
{
        public function checkout($order){}
}
```

Всё кажется вполне логичным и закономерным. Но есть одна проблема — класс `Customer` зависит от класса `OrderProcessor` (мало того, не выполняется и принцип открытости/закрытости).
Для того, чтобы избавится от зависимости от конкретного класса, надо сделать так чтобы `Customer` зависел от абстракции, т.е. от интерфейса `IOrderProcessor`. Данную зависимость можно внедрить через сеттеры, параметры метода, или `Dependency Injection` контейнера. Я решил остановится на 2 методе и получил следующий код.

Инвертирование зависимости класса Customer

```
class Customer
{
        private $currentOrder = null;

        public function buyItems(IOrderProcessor
$processor)
        {
                if(is_null($this->currentOrder)){
                        return false;
                }

                return
$processor->checkout($this->currentOrder);
```

```
        }

        public function addItem($item){
                if(is_null($this->currentOrder)){
                        $this->currentOrder = new
Order();
                }
                return
$this->currentOrder->addItem($item);
        }
        public function deleteItem($item){
                if(is_null($this->currentOrder)){
                        return false;
                }
                return $this->currentOrder
->deleteItem($item);
        }
}

interface IOrderProcessor
{
        public function checkout($order);
}

class OrderProcessor implements IOrderProcessor
{
        public function checkout($order){}
}
```

Таким образом, класс `Customer` теперь зависит только от абстракции, а конкретную реализацию, т.е. детали, ему не так важны.

Шпаргалка

Резюмируя всё выше изложенное, хотелось бы сделать следующую шпаргалку

- **Принцип единственной ответственности (Single responsibility)**
  «На каждый объект должна быть возложена одна единственная обязанность»
  Для этого проверяем, сколько у нас есть причин для изменения класса — если больше одной, то следует разбить данный класс.
- **Принцип открытости/закрытости (Open-closed)**
  «Программные сущности должны быть открыты для расширения, но закрыты для модификации»
  Для этого представляем наш класс как «чёрный ящик» и смотрим, можем ли в таком случае изменить его поведение.
- **Принцип подстановки Барбары Лисков (Liskov substitution)**
  «Объекты в программе могут быть заменены их наследниками без изменения свойств программы»
  Для этого проверяем, не усилили ли мы предусловия и не ослабили ли постусловия. Если это произошло — то принцип не соблюдается
- **Принцип разделения интерфейса (Interface segregation)**
  «Много специализированных интерфейсов лучше, чем один универсальный»
  Проверяем, насколько много интерфейс содержит методов и насколько разные функции накладываются на эти методы, и если необходимо — разбиваем интерфейсы.
- **Принцип инверсии зависимостей (Dependency Invertion)**
  «Зависимости должны строится относительно абстракций, а не деталей»
  Проверяем, зависят ли классы от каких-то других классов(непосредственно инстанцируют объекты других классов и т.д) и если эта зависимость имеет место, заменяем на зависимость от абстракции.

Надеюсь, моя «шпаргалка» поможет кому-нибудь в понимании принципов SOLID и даст толчок к их использованию в своих проектах.
Спасибо за внимание.

P.S. В комментариях посоветовали хорошую книгу — Роберт Мартин «Быстрая разработка программ». Там очень подробно и с примерами описаны принципы SOLID.