Шаблоны проектирования в **PHP**

Павел

3 года назад

Что же такое шаблоны проектирования? Шаблоны проектирования это не шаблоны анализа, это не описания стандартных структур (например, связанных списков). Это не определенные разработки приложений или фреймворков. По сути, шаблоны проектирования это «описания взаимодействующих объектов и классов, предназначенных для решения общей проблемы проектирования в определенном контексте». Иным словами, шаблоны проектирования предоставляют обобщенное, многократно применяемое решение проблем программирования, с которыми мы сталкиваемся каждый день. Шаблоны проектирования это не готовые классы или библиотеки, которые можно просто применить к вашей системе. Это не конкретное решение, которое можно преобразовать в исходный код. Шаблоны проектирования – это намного больше. Это шаблоны, которые можно использовать для решения проблемы в различных конкретных ситуациях.

Шаблоны проектирования помогают ускорить разработку, потому что они уже проверены временем, и со стороны разработчика требуется лишь их выполнить. Шаблоны проектирования не только ускоряют процесс разработки программного обеспечения, но и представляют сложные понятия в более простой форме. Однако нужно быть острожным и не использовать шаблоны проектирования там, где это не нужно. Помимо теории, мы также приведем наиболее абстрактные и простые примеры шаблонов проектирования.

> «Каждый шаблон описывает проблему, которая возникает снова и снова … а затем описывает суть решения этой проблемы, причем таким образом, что вы можете применять это решение миллионы раз, ни разу не повторившись.” – Кристофер Александр

На данный момент существует 23 шаблона проектирования, которые по их назначению можно разделить на три категории:

- **Порождающие шаблоны**: используются для создания объектов, которые можно отделять от их системы реализации
- **Структурные шаблоны**: используются для формирования больших объектных

структур между множеством разрозненных объектов

- **Поведенческие шаблоны**: используются для управления алгоритмами, отношениями и обязанностями между объектами

Ниже приведен полный список шаблонов проектирования:

Порождающие шаблоны

- **Abstract Factory** (Абстрактная фабрика) - позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение.
- **Builder** (Строитель) - используется для отделения процесса конструирования сложного объекта от его представления, так что в результате одного и того же конструирования могут получаться различные объекты. Этот паттерн очень похож на абстрактную фабрику, но в нем акцентируется пошаговое конструирование объекта - в отличие от фабрики, где конструируется семейство классов.
- **Factory Method** (Фабричный метод) - предоставляет подклассам интерфейс для создания экземпляров некоторого класса.
- **Prototype** (Прототип) - используется для задания вида создаваемых объектов на основе объекта прототипа, от которого происходит передача внутреннего состояния (создаёт новые объекты путём копирования прототипа).

Структурные шаблоны

- **Adapter** (Адаптер) - предназначен для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.
- **Bridge** (Мост) - используется для отделения абстракции от ее реализации так, чтобы и то и другое можно было изменять независимо.
- **Composite** (Компоновщик) - используется для компоновки объектов в древовидные структуры для представления иерархий, позволяя одинаково трактовать индивидуальные и составные объекты.
- **Decorator** (Декоратор) - используется для динамического расширения функциональности объекта. Является гибкой альтернативой наследованию.
- **Facade** (Фасад) - представляет собой унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Паттерн фасад определяет интерфейс более высокого уровня, который упрощает использование подсистем.
- **Flyweight** (Приспособленец) - используется для уменьшения затрат при работе с большим количеством мелких объектов.
- **Proxy** (Прокси) - который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

Поведенческие шаблоны

- **Chain of responsibility** (Цепочка обязанностей) - служит для ослабления связи между отправителем и получателем запроса. При этом сам по себе запрос может быть произвольным.
- **Command** (Команда) - представляет собой действие. Объект команды заключает в себе само действие и его параметры.
- **Interpreter** (Интерпретатор) - решает часто встречающуюся, но подверженную изменениям, задачу.
- **Iterator** (Итератор) - представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.
- **Mediator** (Медиатор) - обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
- **Memento** (Хранитель) - позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в это состояние.
- **Observer** (Наблюдатель) - создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.
- **State** (Состояние) - используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.
- **Strategy** (Стратегия) - предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.
- **Template Method** (Шаблонный метод) - определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
- **Visitor** (Посетитель) - описывает операцию, которая должна быть выполнена над каждым объектом из некоторой произвольной структуры.

Теперь рассмотрим некоторые из этих шаблонов. Посмотреть все паттерны с примерами и объяснениями на русском вы также сможете по ссылке http://designpatternsphp.readthedocs.org/ru/latest/

Стратегия

Этот шаблон основан на алгоритмах. Вы инкапсулируете определенные виды алгоритмов, не позволяя классу клиента, ответственного за инстанциирование определенного алгоритма, узнать о фактической реализации. Пример:

```php
<?php

interface OutputInterface
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
{
    public function load()
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

Декоратор

Этот шаблон позволят добавлять новое или дополнительное поведение объекту во время выполнения, в зависимости от ситуации. Пример:

```php
<?php

class HtmlTemplate
{
    // any parent class methods
}

class Template1 extends HtmlTemplate
{
```

```
    protected $html;

    public function __construct()
    {
        $this->html = "<p>__text__</p>";
    }

    public function set($html)
    {
        $this->html = $html;
    }

    public function render()
    {
        echo $this->html;
    }
}

class Template2 extends HtmlTemplate
{
    protected $element;

    public function __construct($s)
    {
        $this->element = $s;
        $this->set("<h2>" . $this->_html . "</h2>");
    }

    public function __call($name, $args)
    {
        $this->element->$name($args[0]);
    }
}

class Template3 extends HtmlTemplate
{
    protected $element;

    public function __construct($s)
    {
        $this->element = $s;
        $this->set("<u>" . $this->_html . "</u>");
    }

    public function __call($name, $args)
    {
        $this->element->$name($args[0]);
    }
}
```

Реестр

Этот шаблон немного выделяется из общего списка, потому что он не является Порождающим шаблоном. Реестр это хэш, получить доступ к данным которого можно при помощи статических методов:

```php
<?php

class Package
{
    protected static $data = array();

    public static function set($key, $value)
    {
        self::$data[$key] = $value;
    }

    public static function get($key)
    {
        return isset(self::$data[$key]) ? self::$data[$key] : null;
    }

    final public static function removeObject($key)
    {
        if (array_key_exists($key, self::$data)) {
            unset(self::$data[$key]);
        }
    }
}

Package::set('name', 'Package name');
print_r(Package::get('name'));
// Package name
```

Фабрика

Это еще один очень популярный шаблон. Его название говорит само за себя: это класс, который подобен настоящей фабрике экземпляров объекта. Иными словами, предположим, что мы знаем, что существуют фабрики, производящие определенный тип продукта. Нам не важно, как фабрика производит этот продукт, но мы знаем, что для каждой фабрики есть один универсальный способ запросить этот продукт:

```php
<?php

interface Factory
{
    public function getProduct();
}

interface Product
{
    public function getName();
```

```
}

class FirstFactory implements Factory
{
    public function getProduct()
    {
        return new FirstProduct();
    }
}

class SecondFactory implements Factory
{
    public function getProduct()
    {
        return new SecondProduct();
    }
}

class FirstProduct implements Product
{
    public function getName()
    {
        return 'The first product';
    }
}

class SecondProduct implements Product
{
    public function getName()
    {
        return 'Second product';
    }
}

$factory = new FirstFactory();
$firstProduct = $factory->getProduct();
$factory = new SecondFactory();
$secondProduct = $factory->getProduct();

print_r($firstProduct->getName());

// The first product
print_r($secondProduct->getName());

// Second product
```

Абстрактная фабрика

Есть ситуации, когда у нас имеются фабрики одинакового типа, и мы хотим инкапсулировать логику выбора, какую из фабрик использовать для выполнения определенной задачи. В данном случае на помощь приходит этот шаблон:

```php
<?php

class Config {
    public static $factory = 1;
}

interface Product {
    public function getName();
}

abstract class AbstractFactory
{
    public static function getFactory()
    {
        switch (Config::$factory)
        {
            case 1:
                return new FirstFactory();
            case 2:
                return new SecondFactory();
        }

        throw new Exception('Bad config');
    }

    abstract public function getProduct();
}

class FirstFactory extends AbstractFactory
{
    public function getProduct()
    {
        return new FirstProduct();
    }
}

class FirstProduct implements Product
{
    public function getName()
    {
        return 'The product from the first factory';
    }
}

class SecondFactory extends AbstractFactory
{
    public function getProduct()
    {
        return new SecondProduct();
    }
}

class SecondProduct implements Product
```

```
{
    public function getName()
    {
        return 'The product from second factory';
    }
}

$firstProduct = AbstractFactory::getFactory()->getProduct();

Config::$factory = 2;
$secondProduct = AbstractFactory::getFactory()->getProduct();
print_r($firstProduct->getName());

// The first product from the first factory
print_r($secondProduct->getName());

// Second product from second factory
```

Наблюдатель

За объектом устанавливается наблюдение путем добавления метода, который позволяет зарегистрироваться другому объекту (наблюдателю). Если наблюдаемый объект изменяется, он посылает сообщение объектам, которые были зарегистрированы в качестве наблюдателей:

```
<?php

interface Observer
{
  function onChanged($sender, $args);
}

interface Observable
{
  function addObserver($observer);
}

class CustomerList implements Observable
{
  private $_observers = [];

  public function addCustomer($name)
  {
    foreach($this->_observers as $obs) {
        $obs->onChanged($this, $name);
    }
  }

  public function addObserver($observer)
  {
    $this->_observers []= $observer;
```

```
  }
}

class CustomerListLogger implements Observer
{
    public function onChanged($sender, $args)
    {
        echo( "'$args' Customer has been added to the list \n" );
    }
}

$ul = new CustomerList();
$ul->addObserver( new CustomerListLogger() );
$ul->addCustomer( "Jack" );
```

Адаптер

Этот шаблон позволяет вам создавать класс с новым интерфейсом, с тем, чтобы этот класс мог использоваться системой с определенными методами вызова:

```
<?php

class SimpleBook
{
    private $author;
    private $title;

    public function __construct($author, $title)
    {
        $this->author = $author;
        $this->title  = $title;
    }

    public function getAuthor()
    {
        return $this->author;
    }

    public function getTitle()
    {
        return $this->title;
    }
}

class BookAdapter
{
    private $book;

    public function __construct(SimpleBook $book)
    {
        $this->book = $book;
    }
```

```
    public function getAuthorAndTitle()
    {
        return $this->book->getTitle().' by '.$this->book->getAuthor();
    }
}

// Usage
$book = new SimpleBook("Gamma, Helm, Johnson, and Vlissides", "Design Pattern
$bookAdapter = new BookAdapter($book);
echo 'Author and Title: '.$bookAdapter->getAuthorAndTitle();
```

Отложенная инициализация

Вот еще одна интересная ситуация. Представьте, что у вас есть фабрика, но вы не знаете, какие из ее функций вам нужны, а какие − нет. В таком случае необходимые операции выполняются только если они необходимы и только один раз:

```
<?php

interface Product
{
    public function getName();
}

class Factory
{
    protected $firstProduct;
    protected $secondProduct;

    public function getFirstProduct()
    {
        if (!$this->firstProduct)
        {
            $this->firstProduct = new FirstProduct();
        }

        return $this->firstProduct;
    }

    public function getSecondProduct()
    {
        if (!$this->secondProduct)
        {
            $this->secondProduct = new SecondProduct();
        }

        return $this->secondProduct;
    }
}
```

```php
class FirstProduct implements Product
{
    public function getName()
    {
        return 'The first product';
    }
}

class SecondProduct implements Product
{
    public function getName()
    {
        return 'Second product';
    }
}

$factory = new Factory();
print_r($factory->getFirstProduct()->getName());
// The first product
print_r($factory->getSecondProduct()->getName());
// Second product
print_r($factory->getFirstProduct()->getName());
// The first product
```

Цепочка команд

Этот шаблон также известен под названием Chain of Command. Это цепочка команд с рядом обработчиков. Сообщение (запрос) проходит через ряд этих обработчиков, и на каждом этапе принимается решение о том, может ли данный обработчик обработать запрос или нет. Процесс останавливается тогда, когда выясняется, что обработчик может обработать запрос:

```php
<?php
interface Command
{
    public function onCommand($name, $args);
}

class CommandChain {
    private $_commands = [];

    public function addCommand($cmd)
    {
        $this->_commands[]= $cmd;
    }

    public function runCommand($name, $args)
    {
        foreach($this->_commands as $cmd)
        {
            if ($cmd->onCommand($name, $args)) {
```

```
                return;
            }
        }
    }
}

class CustCommand implements Command
{
    public function onCommand($name, $args)
    {
        if ($name != 'addCustomer')
            return false;

        echo("This is CustomerCommand handling 'addCustomer'\n");
        return true;
    }
}

class MailCommand implements Command
{
    public function onCommand($name, $args)
    {
        if ($name != 'mail')
            return false;

        echo("This is MailCommand handling 'mail'\n");
        return true;
    }
}

$cc = new CommandChain();
$cc->addCommand( new CustCommand());
$cc->addCommand( new MailCommand());
$cc->runCommand('addCustomer', null);
$cc->runCommand('mail', null);
```

Пул объектов

Пул объектов это хэш, в который можно помещать инициализированные объекты и, при необходимости, доставать их оттуда:

```
<?php

class Product
{
    protected $id;

    public function __construct($id)
    {
        $this->id = $id;
    }
```

```
    public function getId()
    {
        return $this->id;
    }
}

class Factory
{
    protected static $products = array();

    public static function pushProduct(Product $product)
    {
        self::$products[$product->getId()] = $product;
    }

    public static function getProduct($id)
    {
        return isset(self::$products[$id]) ? self::$products[$id] : null;
    }

    public static function removeProduct($id)
    {
        if (array_key_exists($id, self::$products))
        {
            unset(self::$products[$id]);
        }
    }
}

Factory::pushProduct(new Product('first'));
Factory::pushProduct(new Product('second'));
print_r(Factory::getProduct('first')->getId());
// first
print_r(Factory::getProduct('second')->getId());
// second
```

Прототип

Иногда некоторые объекты следует инициализировать более одного раза. Поэтому следует сохранить их инициализацию, особенно если она требует много времени и ресурсов. Прототип это заранее инициализированный и сохраненный объект. При необходимости, его можно клонировать:

```
<?php

interface Product
{
}

class Factory
{
```

```
    private $product;

    public function __construct(Product $product)
    {
        $this->product = $product;
    }

    public function getProduct()
    {
        return clone $this->product;
    }
}

class SomeProduct implements Product
{
    public $name;
}

$prototypeFactory = new Factory(new SomeProduct());
$firstProduct = $prototypeFactory->getProduct();
$firstProduct->name = 'The first product';
$secondProduct = $prototypeFactory->getProduct();
$secondProduct->name = 'Second product';
print_r($firstProduct->name);
// The first product
print_r($secondProduct->name);
// Second product
```

Строитель

Этот шаблон используется, если мы хотим инкапсулировать создание сложного объекта:

```
<?php

class Product
{
    private $name;

    public function setName($name)
    {
        $this->name = $name;
    }

    public function getName()
    {
        return $this->name;
    }
}

abstract class Builder
{
    protected $product;
```

```
    final public function getProduct()
    {
        return $this->product;
    }

    public function buildProduct()
    {
        $this->product = new Product();
    }
}

class FirstBuilder extends Builder
{
    public function buildProduct()
    {
        parent::buildProduct();
        $this->product->setName('The product of the first builder');
    }
}

class SecondBuilder extends Builder
{
    public function buildProduct()
    {
        parent::buildProduct();
        $this->product->setName('The product of second builder');
    }
}

class Factory {
    private $builder;

    public function __construct(Builder $builder)
    {
        $this->builder = $builder;
        $this->builder->buildProduct();
    }

    public function getProduct()
    {
        return $this->builder->getProduct();
    }
}

$firstDirector = new Factory(new FirstBuilder());
$secondDirector = new Factory(new SecondBuilder());
print_r($firstDirector->getProduct()->getName());
// The product of the first builder
print_r($secondDirector->getProduct()->getName());
// The product of second builder
```

Design Patterns PHP

Павел

Подписаться Подписки 2 Подписчики 10

Похожие статьи

CodeLobster IDE - Бесплатный PHP, HTML, CSS и JavaScript редактор

Stas

Symfony 4: маршрутизация, контроллеры и шаблоны

Павел

PHP 7.1 - 9 новых функций, о которых вам стоит знать

devacademy

Как создать собственный контейнер внедрения зависимостей на PHP

Павел

Путеводитель Symfony: Компонент HTTPKernel

devacademy

Руководство по модульному тестированию. Часть V: имитирующие методы и переопределение конструкторов

Павел

Руководство по модульному тестированию. Часть IV: Имитирующие объекты, методы-заглушки и внедрение зависимости

Павел

Улучшенная сериализация с Symfony

Павел