



Обучение

Тutorials

Шаблоны проектирования по-человечески: 6 порождающих паттернов, которые упростят жизнь

От **Montgomeri** - 10.06.2017 25579  2[Добавить в избранное](#)

Ультра-простое объяснение шаблонов проектирования, известных как порождающие паттерны. Они решают уйму проблем, так почему бы не рассмотреть их детальнее?



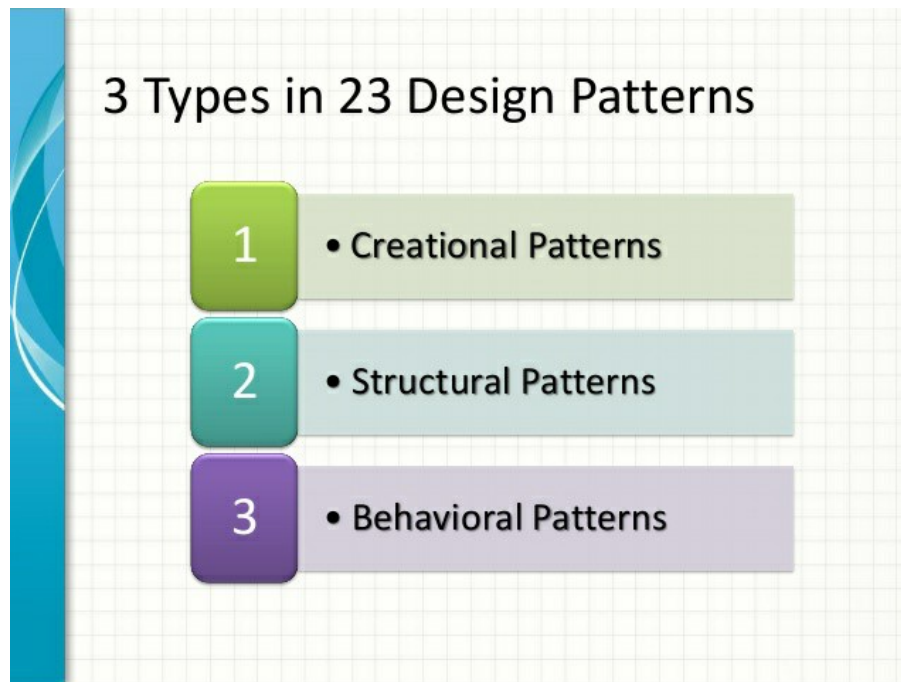
Осторожно!

- шаблоны проектирования – не панацея от всех бед;
- не пытайтесь переусердствовать, в противном случае решение проблем превратится в первопричину этих проблем;
- использовать паттерны проектирования нужно в правильном месте и в правильном порядке.

Основная классификация

Каждый из типов рассчитан на конкретный круг задач, а делятся паттерны на:

1. Порождающие паттерны.
2. Структурные.
3. Поведенческие.



Порождающие паттерны

Этот тип особенно важен, когда система зависит не столько от наследования классов, сколько от композиции. Порождающие паттерны отвечают за создание объектов и позволяют системе быть независимой от типов этих самых объектов и от процесса порождения.

В свою очередь, порождающие паттерны делятся на:

1. Simple Factory
2. Factory Method
3. Abstract Factory
4. Builder
5. Prototype
6. Singleton

1. Паттерн Simple Factory

Предположим, вы строите дом, и вам необходим проход. Было бы глупо всякий раз, когда нужна дверь, облачаться в одежду плотника, чтобы мастерить ее. Вместо этого вы получаете дверь с «завода».

Паттерн предназначен для инкапсуляции процесса образования объектов с помощью отдельного класса. «Простая Фабрика» удобна, но за простоту приходится платить: привязка к конкретной реализации исключает гибкость системы. Simple Factory следует использовать только там, где

архитектура не будет изменяться.

Допустим, у нас есть интерфейс двери:

```
1 interface Door
2 {
3     public function getWidth(): float;
4     public function getHeight(): float;
5 }
6
7 class WoodenDoor implements Door
8 {
9     protected $width;
10    protected $height;
11
12    public function __construct(float $width, float $height)
13    {
14        $this->width = $width;
15        $this->height = $height;
16    }
17
18    public function getWidth(): float
19    {
20        return $this->width;
21    }
22
23    public function getHeight(): float
24    {
25        return $this->height;
26    }
27 }
```

Далее появляется завод, который изготавливает дверь и возвращает ее нам:

```
1 class DoorFactory
2 {
3     public static function makeDoor($width, $height): Door
4     {
5         return new WoodenDoor($width, $height);
6     }
7 }
```

И только после этого мы можем воспользоваться нашей дверью:

```
1 $door = DoorFactory::makeDoor(100, 200);
2 echo 'Ширина: ' . $door->getWidth();
3 echo 'Высота: ' . $door->getHeight();
```

2. Паттерн Factory Method

Но порождающие паттерны на этом не заканчиваются. Шаблон проектирования Factory Method работает с полиморфизмом. В главном классе задается интерфейс, а реализация определяется уже подклассами.

Допустим, у нас есть интерфейс соискателя:

```
1 interface Interviewer
2 {
3     public function askQuestions();
4 }
5
6 class Developer implements Interviewer
7 {
```

```

8     public function askQuestions()
9     {
10         echo 'Спросить о шаблонах проектирования';
11     }
12 }
13
14 class CommunityExecutive implements Interviewer
15 {
16     public function askQuestions()
17     {
18         echo 'Спросить об общественном строительстве';
19     }
20 }

```

Теперь создаем менеджера по подбору персонала:

```

1 abstract class HiringManager
2 {
3
4     // Factory method
5     abstract public function makeInterviewer(): Interviewer;
6
7     public function takeInterview()
8     {
9         $interviewer = $this->makeInterviewer();
10        $interviewer->askQuestions();
11    }
12 }

```

Предоставляем необходимого соискателя:

```

1 class DevelopmentManager extends HiringManager
2 {
3     public function makeInterviewer(): Interviewer
4     {
5         return new Developer();
6     }
7 }
8
9 class MarketingManager extends HiringManager
10 {
11     public function makeInterviewer(): Interviewer
12     {
13         return new CommunityExecutive();
14     }
15 }

```

После чего можно использовать:

```

1 $devManager = new DevelopmentManager();
2 $devManager->takeInterview();
3
4 $marketingManager = new MarketingManager();
5 $marketingManager->takeInterview();

```

3. Паттерн Abstract Factory

Вернемся к примеру из Simple Factory. Может понадобится деревянная дверь, металлическая или пластиковая. Разные типы дверей поставляются из разных магазинов, да и специалисты должны быть соответствующие: плотник, сварщик и т. д. Нам нужна «Абстрактная Фабрика», которая объединяет разные, но связанные фабрики без указания их конкретных классов.

Есть интерфейс двери и некоторые этапы реализации для нее :

```

1 interface Door
2 {
3     public function getDescription();
4 }
5
6 class WoodenDoor implements Door
7 {
8     public function getDescription()
9     {
10         echo 'Я деревянная дверь';
11     }
12 }
13
14 class IronDoor implements Door
15 {
16     public function getDescription()
17     {
18         echo 'Я железная дверь';
19     }
20 }

```

Получаем экспертов для каждого типа дверей:

```

1 interface DoorFittingExpert
2 {
3     public function getDescription();
4 }
5
6 class Welder implements DoorFittingExpert
7 {
8     public function getDescription()
9     {
10         echo 'Я могу подобрать только железные двери';
11     }
12 }
13
14 class Carpenter implements DoorFittingExpert
15 {
16     public function getDescription()
17     {
18         echo 'Я могу подобрать только деревянные двери';
19     }
20 }

```

Имеем ту самую Abstract Factory для создания семейства объектов:

```

1 interface DoorFactory
2 {
3     public function makeDoor(): Door;
4     public function makeFittingExpert(): DoorFittingExpert;
5 }
6
7 // Завод по работе с деревом и плотник
8 class WoodenDoorFactory implements DoorFactory
9 {
10     public function makeDoor(): Door
11     {
12         return new WoodenDoor();
13     }
14
15     public function makeFittingExpert(): DoorFittingExpert
16     {
17         return new Carpenter();
18     }
19 }
20
21 // Завод по производству железных дверей и сварщик
22 class IronDoorFactory implements DoorFactory
23 {
24     public function makeDoor(): Door
25     {
26         return new IronDoor();

```

```

27     }
28
29     public function makeFittingExpert(): DoorFittingExpert
30     {
31         return new Welder();
32     }
33 }

```

Используем:

```

1  $woodenFactory = new WoodenDoorFactory();
2
3  $door = $woodenFactory->makeDoor();
4  $expert = $woodenFactory->makeFittingExpert();
5
6  $door->getDescription(); // На выходе: я деревянная дверь
7  $expert->getDescription(); // На выходе: я могу установить только деревянную дверь
8
9  // То же для завода по изготовлению железных дверей
10 $ironFactory = new IronDoorFactory();
11
12 $door = $ironFactory->makeDoor();
13 $expert = $ironFactory->makeFittingExpert();
14
15 $door->getDescription(); // На выходе: я железная дверь
16 $expert->getDescription(); // На выходе: я могу установить только железную дверь

```

4. Паттерн Builder

Вы зашли в ресторан быстрого питания, заказали гамбургер, и вам его приготовили. Но представьте, что нужно индивидуальное приготовление с определенным хлебом, соусом, сыром и т. д. Здесь поможет шаблон «Строитель», который отвечает за процесс поэтапного создания объекта.

У нас есть желаемый гамбургер:

```

1  class Burger
2  {
3      protected $size;
4
5      protected $cheese = false;
6      protected $pepperoni = false;
7      protected $lettuce = false;
8      protected $tomato = false;
9
10     public function __construct(BurgerBuilder $builder)
11     {
12         $this->size = $builder->size;
13         $this->cheese = $builder->cheese;
14         $this->pepperoni = $builder->pepperoni;
15         $this->lettuce = $builder->lettuce;
16         $this->tomato = $builder->tomato;
17     }
18 }

```

Применяем конструирование:

```

1  class BurgerBuilder
2  {
3      public $size;
4
5      public $cheese = false;
6      public $pepperoni = false;
7      public $lettuce = false;
8      public $tomato = false;
9
10     public function __construct(int $size)

```

```

11     {
12         $this->size = $size;
13     }
14
15     public function addPepperoni()
16     {
17         $this->pepperoni = true;
18         return $this;
19     }
20
21     public function addLettuce()
22     {
23         $this->lettuce = true;
24         return $this;
25     }
26
27     public function addCheese()
28     {
29         $this->cheese = true;
30         return $this;
31     }
32
33     public function addTomato()
34     {
35         $this->tomato = true;
36         return $this;
37     }
38
39     public function build(): Burger
40     {
41         return new Burger($this);
42     }
43 }

```

Используем:

```

1 $burger = (new BurgerBuilder(14))
2           ->addPepperoni()
3           ->addLettuce()
4           ->addTomato()
5           ->build();

```

5. Паттерн Prototype

Помните Долли? Овцу, которую клонировали. Порождающие паттерны «Прототип» – это именно о клонировании.

В PHP это легко реализовать, используя clone:

```

1 class Sheep
2 {
3     protected $name;
4     protected $category;
5
6     public function __construct(string $name, string $category = 'Горная овца')
7     {
8         $this->name = $name;
9         $this->category = $category;
10    }
11
12    public function setName(string $name)
13    {
14        $this->name = $name;
15    }
16
17    public function getName()
18    {

```

```

19     return $this->name;
20 }
21
22 public function setCategory(string $category)
23 {
24     $this->category = $category;
25 }
26
27 public function getCategory()
28 {
29     return $this->category;
30 }
31 }

```

После можно приступать к клонированию, как показано ниже:

```

1 $original = new Sheep('Джолли');
2 echo $original->getName(); // Джолли
3 echo $original->getCategory(); // Горная овца
4
5 // Clone and modify what is required
6 $cloned = clone $original;
7 $cloned->setName('Долли');
8 echo $cloned->getName(); // Долли
9 echo $cloned->getCategory(); // Горная овца

```

6. Паттерн Singleton

Президент может быть только один, и именно он привлекается к действию, когда это необходимо. Президент – одиночка, и по аналогии наш паттерн контролирует создание лишь одного экземпляра класса.

Сделайте конструктор закрытым, отключите клонирование, расширение и создайте статическую переменную для экземпляра:

```

1 final class President
2 {
3     private static $instance;
4
5     private function __construct()
6     {
7         // Прячем конструктор
8     }
9
10    public static function getInstance(): President
11    {
12        if (!self::$instance) {
13            self::$instance = new self();
14        }
15
16        return self::$instance;
17    }
18
19    private function __clone()
20    {
21        // Отключаем клонирование
22    }
23
24    private function __wakeup()
25    {
26        // Отключаем расширение
27    }
28 }

```

Используем:


```
1 $president1 = President::getInstance();
2 $president2 = President::getInstance();
3
4 var_dump($president1 === $president2); // истина
```

Также рекомендуем Вам посмотреть:

[Шаблоны проектирования по-человечески: структурные паттерны](#)

[Шаблоны проектирования по-человечески: поведенческие паттерны в примерах](#)

[Лучший видеокурс по шаблонам проектирования](#)

[4 лучших книг о шаблонах проектирования](#)

[20 полезных навыков, которые можно освоить за 3 дня](#)

Хотите получать больше интересных материалов с доставкой?

Подпишитесь на нашу рассылку:

Подписаться

И не беспокойтесь, мы тоже не любим спам. Отписаться можно в любое время.



Читайте наши статьи в Telegram

Теги

Разное

Предыдущая статья

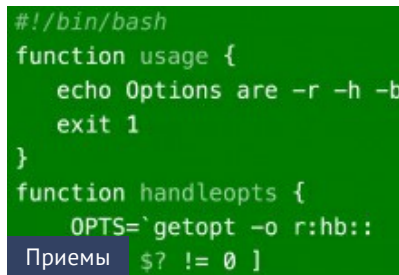
[Об основах алгоритмов сортировки в иллюстрациях](#)

Следующая статья

[Как стать настоящим хакером или Capture The Flag](#)

Похожие статьи

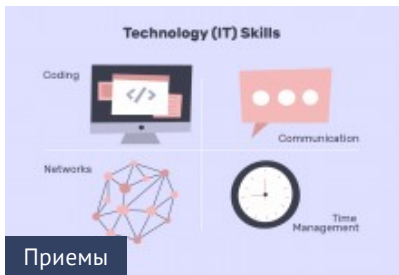
Больше от автора



Как перестать писать плохой код на Bash: практические советы



ТОП-5 историй о том, как IT-консалтинг ускоряет бизнес



10 советов: как подтянуть разговорный английский язык

Комментариев: 2



Николай 06.04.2018, в 17:07

^ 0 v

для приготовления бургера, лучше обойти все в цикле, ведь мы не знаем что закажет каждый посетитель(исправлено):

```
public function __construct(BurgerBuilder $builder)
{
    foreach ($builder as $k=>$item)
    {
        $this->$k = $item;
    }

    return $this;
}
```

Ответить

Комментарий:

Добавить

Свежие вакансии

Разместить вакансию

Plarium Krasnodar

Icons Artist

Краснодар

полный день

Plarium ищет талантливого Художника по иконкам. Наш идеальный кандидат – универсальный командный игрок, который умеет эффективно решать сложные задачи в заданные сроки.

ОТКЛИКНУТЬСЯ НА ВАКАНСИЮ

Lead Programming Manager

Удаленная работа

полный день

Мы ищем Unity-разработчика, который вольется в команду Playgendary и поможет нам создавать новые казуальные игры со своей уникальной стилистикой, а также поможет нам стать сильнее в создании ультра-казуальных игр. Удаленно (любой город) или в офис в Минске/Санкт-Петербурге.

ОТКЛИКНУТЬСЯ НА ВАКАНСИЮ

Plarium Krasnodar

Иллюстратор

Краснодар

полный день

В наш рекламный отдел мы ищем талантливого Иллюстратора, который умеет решать сложные задачи в установленные сроки. Присоединяйся к нам и стань героем геймдева!

ОТКЛИКНУТЬСЯ НА ВАКАНСИЮ

Темы

Android Blockchain C# C++ Data Science Frontend Go Hacking iOS

Случайные статьи



Веб-разработка: итоги 2017 и чего ждать в следующем году



Компьютерные науки или программная инженерия – что выбрать?



Лучшие актуальные шпаргалки по C# на все случаи жизни



С чего начать, чтобы стать айтишником, если вы далеки от IT



Как проверить e-mail в JavaScript, не используя PHP?

Популярные материалы



ТОП-8 трендов web-разработки, обязательных в 2019 году



14 советов, с которыми ты начнёшь мыслить как программист



ТОП-13 крутых идей веб-проектов для прокачки навыков



Хороший, спорный, злой Vue.js: опыт перехода с React



Загрузить больше ▾



```
1000001111
11000011111
0100010000
0010000110
0000010000
0100001110
```

О нас

Библиотека программиста — ваш источник образовательного контента в IT-сфере. Мы публикуем обзоры книг, видеолекции и видеоуроки, дайджесты и образовательные статьи, которые помогут вам улучшить процесс познания в разработке.

Подпишись

ВКонтакте | Telegram | Facebook | Instagram | Яндекс.Дзен

Медиаkit | Пользовательское соглашение | Политика конфиденциальности

Связаться с нами

По вопросам рекламы: matvey@proglib.io

Для обратной связи: hello@proglib.io

123022, Москва, Рочдельская ул., 15, к. 17-18, +7 (995) 114-98-90