РАЗРАБОТКА ВЕБ-САЙТОВ*, ПРОЕКТИРОВАНИЕ И РЕФАКТОРИНГ*, АНАЛИЗ И ПРОЕКТИРОВАНИЕ СИСТЕМ*, PHP*, БЛОГ КОМПАНИИ MAIL.RU GROUP

Шаблоны проектирования с человеческим лицом

ПЕРЕВОД

AloneCoder 10 апреля 2017 в 15:49 146k

Оригинал: Kamran Ahmed

Шаблоны проектирования — это способ решения периодически возникающих проблем. Точнее, это руководства по решению

конкретных проблем. Это не классы, пакеты или библиотеки, которые вы можете вставить в своё приложение и ожидать волшебства.

Как сказано в Википедии:

> В программной инженерии шаблон проектирования приложений — это многократно применяемое решение регулярно возникающей проблемы в рамках определённого контекста архитектуры приложения. Шаблон — это не законченное архитектурное решение, которое можно напрямую преобразовать в исходный или машинный код. Это описание подхода к решению проблемы, который можно применять в разных ситуациях.

⚠️ Будьте осторожны

- Шаблоны проектирования — не «серебряная пуля».
- Не пытайтесь внедрять их принудительно, последствия могут быть негативными. Помните, что шаблоны — это способы **решения**, а не **поиска** проблем. Так что не перемудрите.
- Если применять их правильно и в нужных местах, они могут оказаться спасением. В противном случае у вас будет ещё больше проблем.

В статье приведены примеры на PHP 7, но пусть вас это не смущает, ведь заложенные в шаблонах принципы неизменны. Кроме того, **внедряется поддержка других языков**.

Виды шаблонов проектирования

- Порождающие
- Структурные
- Поведенческие

Порождающие шаблоны проектирования

Вкратце

Порождающие шаблоны описывают создание (instantiate) объекта или группы связанных объектов.

Википедия

> В программной инженерии порождающими называют шаблоны, которые используют механизмы создания объектов, чтобы создавать объекты подходящим для данной ситуации способом. Базовый способ создания может привести к проблемам в архитектуре или к её усложнению. Порождающие шаблоны пытаются решать эти проблемы, управляя способом создания объектов.

- Простая фабрика
- Фабричный метод
- Абстрактная фабрика
- Строитель
- Прототип
- Одиночка

🏠 Простая фабрика

Аналогия

Допустим, вы строите дом и вам нужны двери. Будет бардак, если каждый раз, когда вам требуется дверь, вы станете вооружаться инструментами и делать её на стройплощадке. Вместо этого вы закажете двери на фабрике.

Вкратце

Простая фабрика просто генерирует экземпляр для клиента без предоставления какой-либо логики экземпляра.

Википедия

> В объектно ориентированном программировании фабрикой называется объект, создающий другие объекты. Формально фабрика — это функция или метод, возвращающая объекты разных прототипов или классов из вызова какого-то метода, который считается новым.

Пример

Для начала нам нужен интерфейс двери и его реализация.

```
interface Door
{
    public function getWidth(): float;
    public function getHeight(): float;
}
```

```
class WoodenDoor implements Door
{
    protected $width;
    protected $height;

    public function __construct(float $width, float $height)
    {
        $this->width = $width;
        $this->height = $height;
    }

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

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

Теперь соорудим фабрику дверей, которая создаёт и возвращает нам двери.

```
class DoorFactory
{
    public static function makeDoor($width, $height): Door
    {
        return new WoodenDoor($width, $height);
    }
}
```

Использование:

```
$door = DoorFactory:makeDoor(100, 200);
echo 'Width: ' . $door->getWidth();
echo 'Height: ' . $door->getHeight();
```

Когда использовать?

Когда создание объекта подразумевает какую-то логику, а не просто несколько присваиваний, то имеет смысл делегировать задачу выделенной фабрике, а не повторять повсюду один и тот же код.

🏭 Фабричный метод

Аналогия

Одна кадровичка не в силах провести собеседования со всеми кандидатами на все должности. В зависимости от вакансии она может делегировать разные этапы собеседований разным сотрудникам.

Вкратце

Это способ делегирования логики создания объектов (instantiation logic) дочерним классам.

Википедия

> В классо-ориентированном программировании (class-based programming) фабричным методом называют порождающий шаблон проектирования, использующий генерирующие методы (factory method) для решения проблемы создания объектов без указания для них конкретных классов. Объекты создаются посредством вызова не конструктора, а генерирующего метода,

> определённого в интерфейсе и реализованного дочерними классами либо реализованного в базовом классе и, опционально, переопределённого (overridden) производными классами (derived classes).

Пример

Сначала создадим интерфейс сотрудника, проводящего собеседование, и некоторые реализации для него.

```
interface Interviewer
{
    public function askQuestions();
}

class Developer implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about design patterns!';
    }
}

class CommunityExecutive implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about community building';
    }
}
```

Теперь создадим кадровичку `HiringManager`.

```
abstract class HiringManager
{

    // Фабричный метод
    abstract public function makeInterviewer(): Interviewer;
```

```
    public function takeInterview()
    {
        $interviewer = $this->makeInterviewer();
        $interviewer->askQuestions();
    }
}
```

Любой дочерний класс может расширять его и предоставлять нужного собеседующего:

```
class DevelopmentManager extends HiringManager
{
    public function makeInterviewer(): Interviewer
    {
        return new Developer();
    }
}

class MarketingManager extends HiringManager
{
    public function makeInterviewer(): Interviewer
    {
        return new CommunityExecutive();
    }
}
```

Использование:

```
$devManager = new DevelopmentManager();
$devManager->takeInterview(); // Output: Спрашивает о шаблонах проектирования.

$marketingManager = new MarketingManager();
$marketingManager->takeInterview(); // Output: Спрашивает о создании сообщества.
```

Когда использовать?

Этот шаблон полезен для каких-то общих обработок в классе, но требуемые подклассы динамически определяются в ходе выполнения (runtime). То есть когда клиент не знает, какой именно подкласс может ему понадобиться.

🔨 Абстрактная фабрика

Аналогия

Вернёмся к примеру с дверями из «Простой фабрики». В зависимости от своих потребностей вы можете купить деревянную дверь в одном магазине, стальную — в другом, пластиковую — в третьем. Для монтажа вам понадобятся разные специалисты: деревянной двери нужен плотник, стальной — сварщик, пластиковой — спец по ПВХ-профилям.

Вкратце
Это фабрика фабрик. То есть фабрика, группирующая индивидуальные, но взаимосвязанные/взаимозависимые фабрики без указания для них конкретных классов.

Википедия

> Шаблон «Абстрактная фабрика» описывает способ инкапсулирования группы индивидуальных фабрик, объединённых некой темой, без указания для них конкретных классов.

Пример

Создадим интерфейс Door и несколько реализаций для него.

```
interface Door
{
    public function getDescription();
}

class WoodenDoor implements Door
{
    public function getDescription()
    {
        echo 'I am a wooden door';
    }
}

class IronDoor implements Door
{
    public function getDescription()
    {
        echo 'I am an iron door';
    }
}
```

Теперь нам нужны специалисты по установке каждого вида дверей.

```
interface DoorFittingExpert
{
    public function getDescription();
}

class Welder implements DoorFittingExpert
{
    public function getDescription()
    {
        echo 'I can only fit iron doors';
    }
}

class Carpenter implements DoorFittingExpert
{
    public function getDescription()
```

```
    {
        echo 'I can only fit wooden doors';
    }
}
```

Мы получили абстрактную фабрику, которая позволяет создавать семейства объектов или взаимосвязанные объекты. То есть фабрика деревянных дверей создаст деревянную дверь и человека для её монтажа, фабрика стальных дверей — стальную дверь и соответствующего специалиста и т. д.

```
interface DoorFactory
{
    public function makeDoor(): Door;
    public function makeFittingExpert(): DoorFittingExpert;
}

// Фабрика деревянных дверей возвращает плотника и деревянную дверь
class WoodenDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
        return new WoodenDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Carpenter();
    }
}

// Фабрика стальных дверей возвращает стальную дверь и сварщика
class IronDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
        return new IronDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Welder();
```

```
    }
}
```

Использование:

```
$woodenFactory = new WoodenDoorFactory();

$door = $woodenFactory->makeDoor();
$expert = $woodenFactory->makeFittingExpert();

$door->getDescription();  // Output: Я деревянная дверь
$expert->getDescription(); // Output: Я могу устанавливать только
деревянные двери

// Same for Iron Factory
$ironFactory = new IronDoorFactory();

$door = $ironFactory->makeDoor();
$expert = $ironFactory->makeFittingExpert();

$door->getDescription();  // Output: Я стальная дверь
$expert->getDescription(); // Output: Я могу устанавливать только
стальные двери
```

Здесь фабрика деревянных дверей инкапсулировала `carpenter` и `wooden door`, фабрика стальных дверей — `iron door` and `welder`. То есть можно быть уверенными, что для каждой из созданных дверей мы получим правильного специалиста.

Когда использовать?

Когда у вас есть взаимосвязи с не самой простой логикой создания (creation logic).

👷 Строитель

Аналогия

Допустим, вы пришли в забегаловку, заказали бургер дня, и вам выдали его **без вопросов**. Это пример «Простой фабрики». Но иногда логика создания состоит из большего количества шагов. К примеру, при заказе бургера дня есть несколько вариантов хлеба, начинки, соусов, дополнительных ингредиентов. В таких ситуациях помогает шаблон «Строитель».

Вкратце

Шаблон позволяет создавать разные свойства объекта, избегая загрязнения конструктора (constructor pollution). Это полезно, когда у объекта может быть несколько свойств. Или когда создание объекта состоит из большого количества этапов.

Википедия

> Шаблон «Строитель» предназначен для поиска решения проблемы антипаттерна Telescoping constructor.

Поясню, что такое антипаттерн Telescoping constructor. Каждый из нас когда-либо сталкивался с подобным конструктором:

```
public function __construct($size, $cheese = true, $pepperoni =
true, $tomato = false, $lettuce = true)
{
}
```

Как видите, количество параметров может быстро разрастись, и станет трудно разобраться в их структуре. Кроме того, этот список параметров будет расти и дальше, если в будущем вы захотите добавить новые опции. Это и есть антипаттерн Telescoping constructor.

Пример

Разумная альтернатива — шаблон «Строитель». Сначала создадим бургер:

```
class Burger
{
    protected $size;

    protected $cheese = false;
    protected $pepperoni = false;
    protected $lettuce = false;
    protected $tomato = false;

    public function __construct(BurgerBuilder $builder)
    {
        $this->size = $builder->size;
        $this->cheese = $builder->cheese;
        $this->pepperoni = $builder->pepperoni;
        $this->lettuce = $builder->lettuce;
        $this->tomato = $builder->tomato;
    }
}
```

А затем добавим «строителя»:

```
class BurgerBuilder
{
    public $size;

    public $cheese = false;
```

```
    public $pepperoni = false;
    public $lettuce = false;
    public $tomato = false;

    public function __construct(int $size)
    {
        $this->size = $size;
    }

    public function addPepperoni()
    {
        $this->pepperoni = true;
        return $this;
    }

    public function addLettuce()
    {
        $this->lettuce = true;
        return $this;
    }

    public function addCheese()
    {
        $this->cheese = true;
        return $this;
    }

    public function addTomato()
    {
        $this->tomato = true;
        return $this;
    }

    public function build(): Burger
    {
        return new Burger($this);
    }
}
```

Использование:

```
$burger = (new BurgerBuilder(14))
                    ->addPepperoni()
                    ->addLettuce()
```

```
                    ->addTomato()
                    ->build();
```

Когда использовать?

Когда у объекта может быть несколько свойств и когда нужно избежать Telescoping constructor. Ключевое отличие от шаблона «Простая фабрика»: он используется в одноэтапном создании, а «Строитель» — в многоэтапном.

🐑 Прототип

Аналогия

Помните клонированную овечку Долли? Так вот, этот шаблон проектирования как раз посвящён клонированию.

Вкратце

Объект создаётся посредством клонирования существующего объекта.

Википедия

> Шаблон «Прототип» используется, когда типы создаваемых объектов определяются экземпляром-прототипом, клонированным для создания новых объектов.

То есть шаблон позволяет дублировать существующий объект и модифицировать копию в соответствии с потребностями. Без заморочек с созданием объекта с нуля и его настройкой.

Пример

В PHP это легко можно сделать с помощью `clone`:

```
class Sheep
{
    protected $name;
    protected $category;

    public function __construct(string $name, string $category =
'Mountain Sheep')
    {
        $this->name = $name;
        $this->category = $category;
    }

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

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

    public function setCategory(string $category)
    {
        $this->category = $category;
    }

    public function getCategory()
    {
        return $this->category;
    }
}
```

Затем можно клонировать так:

```
$original = new Sheep('Jolly');
echo $original->getName(); // Джолли
echo $original->getCategory(); // Горная овечка

// Клонируйте и модифицируйте, что нужно
$cloned = clone $original;
$cloned->setName('Dolly');
echo $cloned->getName(); // Долли
echo $cloned->getCategory(); // Горная овечка
```

Также для модификации процедуры клонирования можно обратиться к магическому методу `__clone`.

Когда использовать?

Когда необходимый объект аналогичен уже существующему или когда создание с нуля дороже клонирования.

💍 Одиночка

Аналогия

У страны может быть только один президент. Он должен действовать, когда того требуют обстоятельства и долг. В данном случае президент — одиночка.

Вкратце

Шаблон позволяет удостовериться, что создаваемый объект — единственный в своём классе.

Википедия

> Шаблон «Одиночка» позволяет ограничивать создание класса единственным объектом. Это удобно, когда для координации действий в рамках системы требуется, чтобы объект был единственным в своём классе.

На самом деле шаблон «Одиночка» считается антипаттерном, не следует им слишком увлекаться. Он необязательно плох и иногда бывает полезен. Но применяйте его с осторожностью, потому что «Одиночка» вносит в приложение глобальное состояние, так что изменение в одном месте может повлиять на все остальные случаи использования, а отлаживать такое — не самое простое занятие. Другие недостатки шаблона: он делает ваш код сильно связанным (tightly coupled), а создание прототипа (mocking) «Одиночки» может быть затруднено.

Пример

Сделайте конструктор приватным, отключите расширения и создайте статическую переменную для хранения экземпляра:

```
final class President
{
    private static $instance;

    private function __construct()
    {
        // Прячем конструктор
    }

    public static function getInstance(): President
    {
        if (!self::$instance) {
            self::$instance = new self();
        }
```

```
        return self::$instance;
    }

    private function __clone()
    {
        // Отключаем клонирование
    }

    private function __wakeup()
    {
        // Отключаем десериализацию
    }
}
```

Использование:

```
$president1 = President::getInstance();
$president2 = President::getInstance();

var_dump($president1 === $president2); // true
```

Структурные шаблоны проектирования

Вкратце

Эти шаблоны в основном посвящены компоновке объектов (object composition). То есть тому, как сущности могут друг друга использовать. Ещё одно объяснение: структурные шаблоны помогают ответить на вопрос «Как построить программный компонент?»

Википедия

> Структурными называют шаблоны, которые облегчают проектирование, определяя простой способ реализации

> взаимоотношений между сущностями.

- Адаптер
- Мост
- Компоновщик
- Декоратор
- Фасад
- Приспособленец
- Заместитель

🔌 Адаптер

Аналогия

Допустим, у вас на карте памяти есть какие-то картинки. Их нужно перенести на компьютер. Нужен адаптер, совместимый с входным портом компьютера, в который можно вставить карту памяти. В данном примере адаптер — это картридер. Ещё один пример: переходник, позволяющий использовать американский блок питания с российской розеткой. Третий пример: переводчик — это адаптер, соединяющий двух людей, говорящих на разных языках.

Вкратце

Шаблон «Адаптер» позволяет помещать несовместимый объект в обёртку, чтобы он оказался совместимым с другим классом.

Википедия

> Шаблон проектирования «Адаптер» позволяет использовать интерфейс существующего класса как другой интерфейс. Этот шаблон часто применяется для обеспечения работы одних классов с другими без изменения их исходного кода.

Пример

Представим себе охотника на львов.

Создадим интерфейс `Lion`, который реализует все типы львов.

```
interface Lion
{
    public function roar();
}

class AfricanLion implements Lion
{
    public function roar()
    {
    }
}

class AsianLion implements Lion
{
    public function roar()
    {
    }
}
```

Охотник должен охотиться на все реализации интерфейса `Lion`.

```
class Hunter
{
    public function hunt(Lion $lion)
    {
    }
}
```

Добавим теперь дикую собаку `WildDog`, на которую охотник тоже может охотиться. Но у нас не получится сделать это напрямую, потому что у собаки другой интерфейс. Чтобы она стала совместима с охотником, нужно создать подходящий адаптер.

```
// Это нужно добавить
class WildDog
{
    public function bark()
    {
    }
}

// Адаптер вокруг собаки сделает её совместимой с охотником
class WildDogAdapter implements Lion
{
    protected $dog;

    public function __construct(WildDog $dog)
    {
        $this->dog = $dog;
    }

    public function roar()
    {
        $this->dog->bark();
    }
}
```

Теперь `WildDog` может вступить в игру действие благодаря `WildDogAdapter`.

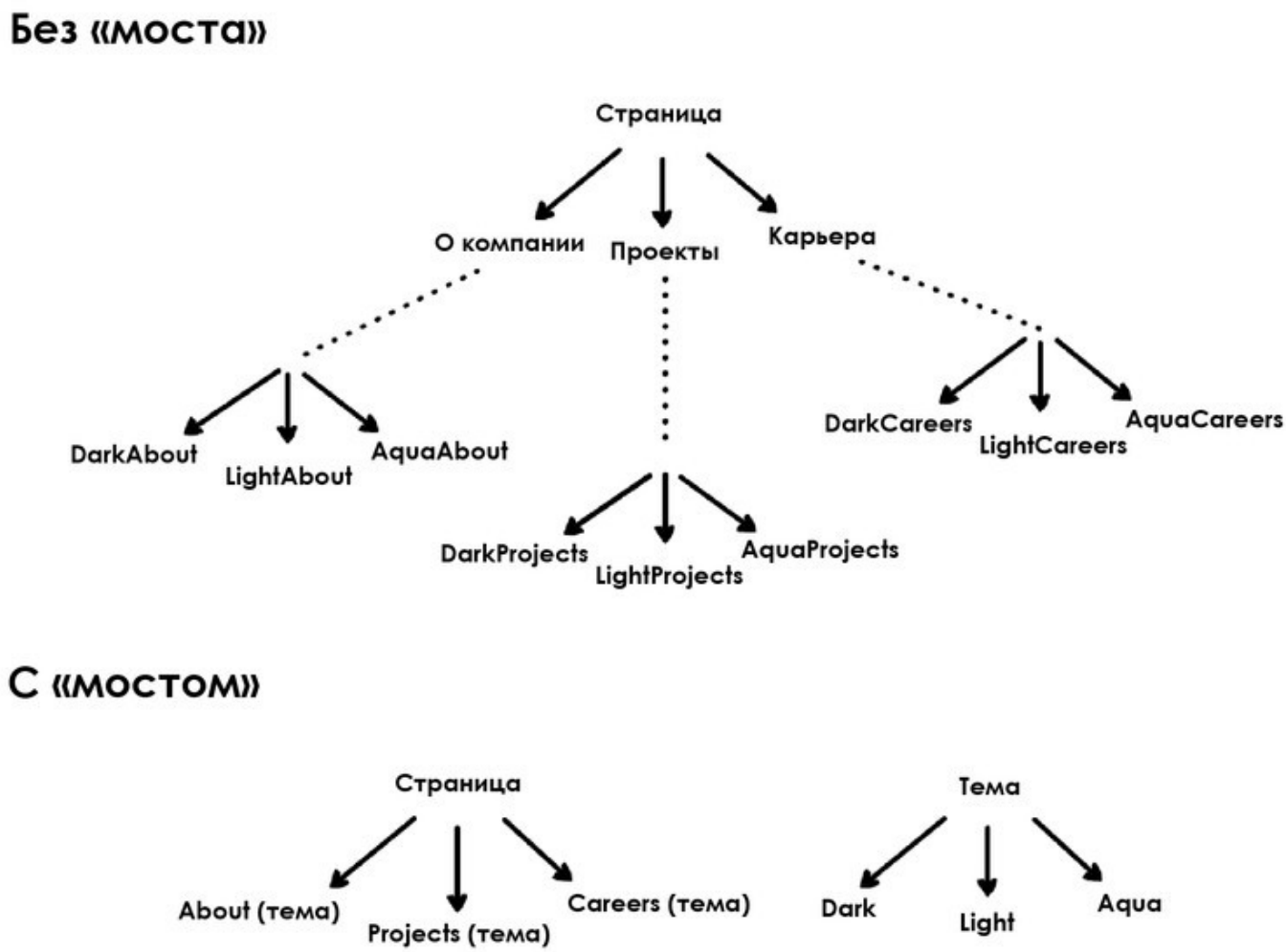
```
$wildDog = new WildDog();
$wildDogAdapter = new WildDogAdapter($wildDog);

$hunter = new Hunter();
$hunter->hunt($wildDogAdapter);
```

Мост

Аналогия

Допустим, у вас есть сайт с несколькими страницами. Вы хотите позволить пользователям менять темы оформления страниц. Как бы вы поступили? Создали множественные копии каждой страницы для каждой темы или просто сделали отдельные темы и подгружали их в соответствии с настройками пользователей? Шаблон «Мост» позволяет реализовать второй подход.



Вкратце

Шаблон «Мост» — это предпочтение компоновки наследованию. Подробности реализации передаются из одной иерархии другому объекту с отдельной иерархией.

Википедия

> Шаблон «Мост» означает отделение абстракции от реализации, чтобы их обе можно было изменять независимо друг от друга.

Пример

Реализуем вышеописанный пример с веб-страницами. Сделаем иерархию `WebPage`:

```
interface WebPage
{
    public function __construct(Theme $theme);
    public function getContent();
}

class About implements WebPage
{
    protected $theme;

    public function __construct(Theme $theme)
    {
        $this->theme = $theme;
    }

    public function getContent()
    {
        return "About page in " . $this->theme->getColor();
    }
}

class Careers implements WebPage
{
    protected $theme;
```

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

    public function getContent()
    {
        return "Careers page in " . $this->theme->getColor();
    }
}
```

Отделим иерархию тем:

```
interface Theme
{
    public function getColor();
}

class DarkTheme implements Theme
{
    public function getColor()
    {
        return 'Dark Black';
    }
}
class LightTheme implements Theme
{
    public function getColor()
    {
        return 'Off white';
    }
}
class AquaTheme implements Theme
{
    public function getColor()
    {
        return 'Light blue';
    }
}
```

Обе иерархии:

```
$darkTheme = new DarkTheme();

$about = new About($darkTheme);
$careers = new Careers($darkTheme);

echo $about->getContent(); // "About page in Dark Black";
echo $careers->getContent(); // "Careers page in Dark Black";
```

🌿 Компоновщик

Аналогия

Каждая компания состоит из сотрудников. У каждого сотрудника есть одни и те же свойства: зарплата, обязанности, отчётность перед кем-то, субординация...

Вкратце

Шаблон «Компоновщик» позволяет клиентам обрабатывать отдельные объекты в едином порядке.

Википедия

> Шаблон «Компоновщик» описывает общий порядок обработки группы объектов, словно это одиночный экземпляр объекта. Суть шаблона — компонование объектов в древовидную структуру для представления иерархии от частного к целому. Шаблон позволяет клиентам одинаково обращаться к отдельным объектам и к группам объектов.

Пример

Вот разные типы сотрудников:

```
interface Employee
{
    public function __construct(string $name, float $salary);
    public function getName(): string;
    public function setSalary(float $salary);
    public function getSalary(): float;
    public function getRoles(): array;
}

class Developer implements Employee
{
    protected $salary;
    protected $name;

    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }

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

    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
    }
}

class Designer implements Employee
{
```

```
    protected $salary;
    protected $name;

    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }

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

    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
    }
}
```

А вот компания, которая состоит из сотрудников разных типов:

```
class Organization
{
    protected $employees;

    public function addEmployee(Employee $employee)
    {
        $this->employees[] = $employee;
    }

    public function getNetSalaries(): float
    {
        $netSalary = 0;
```

```
        foreach ($this->employees as $employee) {
            $netSalary += $employee->getSalary();
        }

        return $netSalary;
    }
}
```

Использование:

```
// Подготовка сотрудников
$john = new Developer('John Doe', 12000);
$jane = new Designer('Jane Doe', 15000);

// Включение их в штат
$organization = new Organization();
$organization->addEmployee($john);
$organization->addEmployee($jane);

echo "Net salaries: " . $organization->getNetSalaries(); // Net
Salaries: 27000
```

Декоратор

Аналогия

Допустим, у вас свой автосервис, оказывающий различные услуги. Как выставлять клиентам счёт? Добавлять последовательно услуги и их стоимость — и в конце концов получится итоговая сумма к оплате. Здесь каждый тип услуги — это «декоратор».

Вкратце

Шаблон «Декоратор» позволяет во время выполнения динамически изменять поведение объекта, обёртывая его в объект

класса «декоратора».

Википедия

> Шаблон «Декоратор» позволяет подключать к объекту дополнительное поведение (статически или динамически), не влияя на поведение других объектов того же класса. Шаблон часто используется для соблюдения принципа единственной обязанности (Single Responsibility Principle), поскольку позволяет разделить функциональность между классами для решения конкретных задач.

Пример

Возьмём в качестве примера кофе. Сначала просто реализуем интерфейс:

```
interface Coffee
{
    public function getCost();
    public function getDescription();
}

class SimpleCoffee implements Coffee
{
    public function getCost()
    {
        return 10;
    }

    public function getDescription()
    {
        return 'Simple coffee';
    }
}
```

Можно сделать код расширяемым, чтобы при необходимости вносить модификации. Добавим «декораторы»:

```
class MilkCoffee implements Coffee
{
    protected $coffee;

    public function __construct(Coffee $coffee)
    {
        $this->coffee = $coffee;
    }

    public function getCost()
    {
        return $this->coffee->getCost() + 2;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', milk';
    }
}

class WhipCoffee implements Coffee
{
    protected $coffee;

    public function __construct(Coffee $coffee)
    {
        $this->coffee = $coffee;
    }

    public function getCost()
    {
        return $this->coffee->getCost() + 5;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', whip';
    }
}

class VanillaCoffee implements Coffee
{
    protected $coffee;
```

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

    public function getCost()
    {
        return $this->coffee->getCost() + 3;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', vanilla';
    }
}
```

Теперь приготовим кофе:

```
$someCoffee = new SimpleCoffee();
echo $someCoffee->getCost(); // 10
echo $someCoffee->getDescription(); // Simple Coffee

$someCoffee = new MilkCoffee($someCoffee);
echo $someCoffee->getCost(); // 12
echo $someCoffee->getDescription(); // Simple Coffee, milk

$someCoffee = new WhipCoffee($someCoffee);
echo $someCoffee->getCost(); // 17
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip

$someCoffee = new VanillaCoffee($someCoffee);
echo $someCoffee->getCost(); // 20
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip,
vanilla
```

📦 Фасад

Аналогия

Как включить компьютер? Вы скажете: «Нажать кнопку включения». Это потому, что вы используете простой интерфейс, предоставляемый компьютером наружу. А внутри него происходит очень много процессов. Простой интерфейс для сложной подсистемы — это фасад.

Вкратце

Шаблон «Фасад» предоставляет упрощённый интерфейс для сложной подсистемы.

Википедия

> «Фасад» — это объект, предоставляющий упрощённый интерфейс для более крупного тела кода, например библиотеки классов.

Пример

Создадим класс computer:

```
class Computer
{
    public function getElectricShock()
    {
        echo "Ouch!";
    }

    public function makeSound()
    {
        echo "Beep beep!";
    }

    public function showLoadingScreen()
    {
        echo "Loading..";
```

```
    }

    public function bam()
    {
        echo "Ready to be used!";
    }

    public function closeEverything()
    {
        echo "Bup bup bup buzzzz!";
    }

    public function sooth()
    {
        echo "Zzzzz";
    }

    public function pullCurrent()
    {
        echo "Haaah!";
    }
}
```

Теперь «фасад»:

```
class ComputerFacade
{
    protected $computer;

    public function __construct(Computer $computer)
    {
        $this->computer = $computer;
    }

    public function turnOn()
    {
        $this->computer->getElectricShock();
        $this->computer->makeSound();
        $this->computer->showLoadingScreen();
        $this->computer->bam();
    }

    public function turnOff()
    {
        $this->computer->closeEverything();
```

```
        $this->computer->pullCurrent();
        $this->computer->sooth();
    }
}
```

Использование:

```
$computer = new ComputerFacade(new Computer());
$computer->turnOn(); // Ouch! Beep beep! Loading.. Ready to be used!
$computer->turnOff(); // Bup bup buzzz! Haah! Zzzzz
```

🌿 Приспособленец

Аналогия

Обычно в заведениях общепита чай заваривают не отдельно для каждого клиента, а сразу в некой крупной ёмкости. Это позволяет экономить ресурсы: газ/электричество, время и т. д. Шаблон «Приспособленец» как раз посвящён общему использованию (sharing).

Вкратце

Шаблон применяется для минимизирования использования памяти или вычислительной стоимости за счёт общего использования как можно большего количества одинаковых объектов.

Википедия

> «Приспособленец» — это объект, минимизирующий использование памяти за счёт общего с другими, такими же

> объектами использования как можно большего объёма данных. Это способ применения многочисленных объектов, когда простое повторяющееся представление приведёт к неприемлемому потреблению памяти.

Пример

Сделаем типы чая и чайника.

```
// Приспособленец – то, что будет закешировано.
// Типы чая здесь – приспособленцы.
class KarakTea
{
}

// Действует как фабрика и экономит чай
class TeaMaker
{
    protected $availableTea = [];

    public function make($preference)
    {
        if (empty($this->availableTea[$preference])) {
            $this->availableTea[$preference] = new KarakTea();
        }

        return $this->availableTea[$preference];
    }
}
```

Сделаем забегаловку `TeaShop`, принимающую и обрабатывающую заказы:

```
class TeaShop
{
    protected $orders;
    protected $teaMaker;

    public function __construct(TeaMaker $teaMaker)
```

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

    public function takeOrder(string $teaType, int $table)
    {
        $this->orders[$table] = $this->teaMaker->make($teaType);
    }

    public function serve()
    {
        foreach ($this->orders as $table => $tea) {
            echo "Serving tea to table# " . $table;
        }
    }
}
```

Использование:

```
$teaMaker = new TeaMaker();
$shop = new TeaShop($teaMaker);

$shop->takeOrder('less sugar', 1);
$shop->takeOrder('more milk', 2);
$shop->takeOrder('without sugar', 5);

$shop->serve();
// Serving tea to table# 1
// Serving tea to table# 2
// Serving tea to table# 5
```

🎱 Заместитель

Аналогия

Открыть дверь с электронным замком можно с помощью карточки доступа (access card) или кнопки для обхода системы безопасности. То есть основная функциональность двери —

открыться, а поверх неё может быть ещё какая-то функциональность — «заместитель».

Вкратце

С помощью шаблона «Заместитель» класс представляет функциональность другого класса.

Википедия

> В наиболее общей форме «Заместитель» — это класс, функционирующий как интерфейс к чему-либо. Это оболочка или объект-агент, вызываемый клиентом для получения доступа к другому, «настоящему» объекту. «Заместитель» может просто переадресовывать запросы настоящему объекту, а может предоставлять дополнительную логику: кеширование данных при интенсивном выполнении операций или потреблении ресурсов настоящим объектом; проверка предварительных условий (preconditions) до вызова выполнения операций настоящим объектом.

Пример

Реализуем интерфейс двери и саму дверь:

```
interface Door
{
    public function open();
    public function close();
}

class LabDoor implements Door
{
```

```
    public function open()
    {
        echo "Opening lab door";
    }

    public function close()
    {
        echo "Closing the lab door";
    }
}
```

Сделаем «заместителя», чтобы дверь могла выполнять защитную функцию:

```
class Security
{
    protected $door;

    public function __construct(Door $door)
    {
        $this->door = $door;
    }

    public function open($password)
    {
        if ($this->authenticate($password)) {
            $this->door->open();
        } else {
            echo "Big no! It ain't possible.";
        }
    }

    public function authenticate($password)
    {
        return $password === '$ecr@t';
    }

    public function close()
    {
        $this->door->close();
    }
}
```

Использование:

```
$door = new Security(new LabDoor());
$door->open('invalid'); // Big no! It ain't possible.

$door->open('$ecr@t'); // Opening lab door
$door->close(); // Closing lab door
```

Ещё один пример связан с реализацией преобразователя данных (data-mapper). С помощью этого шаблона я недавно сделал ODM (Object Data Mapper) для MongoDB. Я написал «заместителя» вокруг mongo-классов, воспользовавшись волшебным методом `__call()`. Все вызовы методов проходили к оригинальным mongo-классам через «заместителя», а извлечённые результаты возвращались как есть. Только в случае с `find` или `findOne` данные преобразовывались в объекты требуемого класса, которые возвращались вместо `Cursor`.

Поведенческие шаблоны проектирования

Вкратце

Они связаны с присвоением обязанностей (responsibilities) объектам. От структурных шаблонов они отличаются тем, что не просто описывают структуру, но и очерчивают шаблоны передачи данных, обеспечения взаимодействия. То есть поведенческие шаблоны позволяют ответить на вопрос «Как реализовать поведение в программном компоненте?»

Википедия

> Поведенческие шаблоны проектирования определяют алгоритмы и способы реализации взаимодействия различных объектов и классов. Они обеспечивают гибкость взаимодействия между объектами.

- Цепочка ответственности
- Команда
- Итератор
- Посредник
- Хранитель
- Наблюдатель
- Посетитель
- Стратегия
- Состояние
- Шаблонный метод

🔗 Цепочка ответственности

Аналогия

Допустим, для вашего банковского счёта доступны три способа оплаты (`A`, `B` и `C`). Каждый подразумевает разные доступные суммы денег: `A` — 100 долларов, `B` — 300, `C` — 1000. Приоритетность способов при оплате: A, затем B, затем C. Вы пытаетесь купить что-то за 210 долларов. На основании «цепочки ответственности» система попытается оплатить способом A. Если денег хватает — то оплата проходит, а цепочка прерывается. Если денег не хватает — то система переходит к способу B, и т. д.

Вкратце

Шаблон «Цепочка ответственности» позволяет создавать цепочки объектов. Запрос входит с одного конца цепочки и движется от объекта к объекту, пока не будет найден подходящий обработчик.

Википедия

> Шаблон «Цепочка ответственности» содержит исходный управляющий объект и ряд обрабатывающих объектов. Каждый обрабатывающий объект содержит логику, определяющую типы командных объектов, которые он может обрабатывать, а остальные передаются по цепочке следующему обрабатывающему объекту.

Пример

Создадим основной банковский счёт, содержащий логику связывания счетов в цепочки, и сами счета.

```
abstract class Account
{
    protected $successor;
    protected $balance;

    public function setNext(Account $account)
    {
        $this->successor = $account;
    }

    public function pay(float $amountToPay)
    {
        if ($this->canPay($amountToPay)) {
            echo sprintf('Paid %s using %s' . PHP_EOL, $amountToPay,
get_called_class());
```

```
        } elseif ($this->successor) {
            echo sprintf('Cannot pay using %s. Proceeding ..' .
PHP_EOL, get_called_class());
            $this->successor->pay($amountToPay);
        } else {
            throw new Exception('None of the accounts have enough
balance');
        }
    }

    public function canPay($amount): bool
    {
        return $this->balance >= $amount;
    }
}

class Bank extends Account
{
    protected $balance;

    public function __construct(float $balance)
    {
        $this->balance = $balance;
    }
}

class Paypal extends Account
{
    protected $balance;

    public function __construct(float $balance)
    {
        $this->balance = $balance;
    }
}

class Bitcoin extends Account
{
    protected $balance;

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

Теперь с помощью определённых выше линков (Bank, Paypal, Bitcoin) подготовим цепочку:

```
// Сделаем такую цепочку
//      $bank->$paypal->$bitcoin
//
// Приоритет у банка
//      Если банк не может оплатить, переходим к Paypal
//      Если Paypal не может, переходим к Bitcoin

$bank = new Bank(100);          // У банка баланс 100
$paypal = new Paypal(200);      // У Paypal баланс 200
$bitcoin = new Bitcoin(300);    // У Bitcoin баланс 300

$bank->setNext($paypal);
$paypal->setNext($bitcoin);

// Начнём с банка
$bank->pay(259);

// Выходной вид
// ==============
// Нельзя оплатить с помощью банка. Обрабатываю...
// Нельзя оплатить с помощью Paypal. Обрабатываю...
// Оплачено 259 с помощью Bitcoin!
```

👮 Команда

Аналогия

Вы пришли в ресторан. Вы (`Client`) просите официанта (`Invoker`) принести блюда (`Command`). Официант перенаправляет запрос шеф-повару (`Receiver`), который знает, что и как готовить. Другой пример: вы (`Client`) включаете (`Command`) телевизор (`Receiver`) с помощью пульта (`Invoker`).

Вкратце

Шаблон «Команда» позволяет инкапсулировать действия в объекты. Ключевая идея — предоставить средства отделения клиента от получателя.

Википедия

> В шаблоне «Команда» объект используется для инкапсуляции всей информации, необходимой для выполнения действия либо для его инициирования позднее. Информация включает в себя имя метода; объект, владеющий методом; значения параметров метода.

Пример

Сначала сделаем получателя, содержащего реализации каждого действия, которое может быть выполнено.

```
// Receiver
class Bulb
{
    public function turnOn()
    {
        echo "Bulb has been lit";
    }

    public function turnOff()
    {
        echo "Darkness!";
    }
}
```

Теперь сделаем интерфейс, который будет реализовывать каждая команда. Также сделаем набор команд.

```
interface Command
{
    public function execute();
    public function undo();
    public function redo();
}

// Command
class TurnOn implements Command
{
    protected $bulb;

    public function __construct(Bulb $bulb)
    {
        $this->bulb = $bulb;
    }

    public function execute()
    {
        $this->bulb->turnOn();
    }

    public function undo()
    {
        $this->bulb->turnOff();
    }

    public function redo()
    {
        $this->execute();
    }
}

class TurnOff implements Command
{
    protected $bulb;

    public function __construct(Bulb $bulb)
    {
        $this->bulb = $bulb;
    }

    public function execute()
    {
        $this->bulb->turnOff();
    }
```

```
    public function undo()
    {
        $this->bulb->turnOn();
    }

    public function redo()
    {
        $this->execute();
    }
}
```

Теперь сделаем вызывающего `Invoker`, с которым будет взаимодействовать клиент для обработки команд.

```
// Invoker
class RemoteControl
{
    public function submit(Command $command)
    {
        $command->execute();
    }
}
```

Посмотрим, как всё это может использовать клиент:

```
$bulb = new Bulb();

$turnOn = new TurnOn($bulb);
$turnOff = new TurnOff($bulb);

$remote = new RemoteControl();
$remote->submit($turnOn); // Лампочка зажглась!
$remote->submit($turnOff); // Темнота!
```

Шаблон «Команда» можно использовать и для реализации системы на основе транзакций. То есть системы, в которой вы

сохраняете историю команд по мере их выполнения. Если последняя команда выполнена успешно, то всё хорошо. В противном случае система итерирует по истории и делает `undo` для всех выполненных команд.

👓 Итератор

Аналогия

Хороший пример — радиоприёмник. Вы начинаете с какой-то радиостанции, а затем перемещаетесь по станциям вперёд/назад. То есть устройство предоставляет интерфейс для итерирования по каналам.

Вкратце

Шаблон — это способ доступа к элементам объекта без раскрытия базового представления.

Википедия

> В этом шаблоне итератор используется для перемещения по контейнеру и обеспечения доступа к элементам контейнера. Шаблон подразумевает отделение алгоритмов от контейнера. В каких-то случаях алгоритмы, специфичные для этого контейнера, не могут быть отделены.

Пример

В PHP довольно легко реализовать этот шаблон с помощью стандартной библиотеки PHP. Сначала создадим радиостанцию `RadioStation`.

```
class RadioStation
{
    protected $frequency;

    public function __construct(float $frequency)
    {
        $this->frequency = $frequency;
    }

    public function getFrequency(): float
    {
        return $this->frequency;
    }
}
```

Теперь создадим итератор:

```
use Countable;
use Iterator;

class StationList implements Countable, Iterator
{
    /** @var RadioStation[] $stations */
    protected $stations = [];

    /** @var int $counter */
    protected $counter;

    public function addStation(RadioStation $station)
    {
        $this->stations[] = $station;
    }

    public function removeStation(RadioStation $toRemove)
    {
        $toRemoveFrequency = $toRemove->getFrequency();
        $this->stations = array_filter($this->stations, function
```

```
(RadioStation $station) use ($toRemoveFrequency) {
            return $station->getFrequency() !== $toRemoveFrequency;
        });
    }

    public function count(): int
    {
        return count($this->stations);
    }

    public function current(): RadioStation
    {
        return $this->stations[$this->counter];
    }

    public function key()
    {
        return $this->counter;
    }

    public function next()
    {
        $this->counter++;
    }

    public function rewind()
    {
        $this->counter = 0;
    }

    public function valid(): bool
    {
        return isset($this->stations[$this->counter]);
    }
}
```

Использование:

```
$stationList = new StationList();

$stationList->addStation(new RadioStation(89));
$stationList->addStation(new RadioStation(101));
$stationList->addStation(new RadioStation(102));
$stationList->addStation(new RadioStation(103.2));
```

```
foreach($stationList as $station) {
    echo $station->getFrequency() . PHP_EOL;
}

$stationList->removeStation(new RadioStation(89)); // Will remove
station 89
```

👽 Посредник

Аналогия

Когда вы говорите с кем-то по мобильнику, то между вами и собеседником находится мобильный оператор. То есть сигнал передаётся через него, а не напрямую. В данном примере оператор — посредник.

Вкратце

Шаблон «Посредник» подразумевает добавление стороннего объекта («посредника») для управления взаимодействием между двумя объектами («коллегами»). Шаблон помогает уменьшить связанность (coupling) классов, общающихся друг с другом, ведь теперь они не должны знать о реализациях своих собеседников.

Википедия

> Шаблон определяет объект, который инкапсулирует способ взаимодействия набора объектов.

Пример

Простейший пример: чат («посредник»), в котором пользователи («коллеги») отправляют друг другу сообщения.

Создадим «посредника»:

```
interface ChatRoomMediator
{
    public function showMessage(User $user, string $message);
}

// Посредник
class ChatRoom implements ChatRoomMediator
{
    public function showMessage(User $user, string $message)
    {
        $time = date('M d, y H:i');
        $sender = $user->getName();

        echo $time . '[' . $sender . ']:' . $message;
    }
}
```

Теперь создадим «коллег»:

```
class User {
    protected $name;
    protected $chatMediator;

    public function __construct(string $name, ChatRoomMediator
$chatMediator) {
        $this->name = $name;
        $this->chatMediator = $chatMediator;
    }

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

    public function send($message) {
        $this->chatMediator->showMessage($this, $message);
```

```
    }
}
```

Использование:

```
$mediator = new ChatRoom();

$john = new User('John Doe', $mediator);
$jane = new User('Jane Doe', $mediator);

$john->send('Hi there!');
$jane->send('Hey!');

// Выходной вид
// Feb 14, 10:58 [John]: Hi there!
// Feb 14, 10:58 [Jane]: Hey!
```

💾 Хранитель

Аналогия

В качестве примера можно привести калькулятор («создатель»), у которого любая последняя выполненная операция сохраняется в памяти («хранитель»), чтобы вы могли снова вызвать её с помощью каких-то кнопок («опекун»).

Вкратце

Шаблон «Хранитель» фиксирует и хранит текущее состояние объекта, чтобы оно легко восстанавливалось.

Википедия

> Шаблон «Хранитель» позволяет восстанавливать объект в его предыдущем состоянии (отмена через откат — undo via rollback).

Обычно шаблон применяется, когда нужно реализовать функциональность отмены операции.

Пример

Текстовый редактор время от времени сохраняет своё состояние, чтобы можно было восстановить текст в каком-то прошлом виде.

Сначала создадим объект «хранитель», в котором можно сохранять состояние редактора.

```
class EditorMemento
{
    protected $content;

    public function __construct(string $content)
    {
        $this->content = $content;
    }

    public function getContent()
    {
        return $this->content;
    }
}
```

Теперь сделаем редактор («создатель»), который будет использовать объект «хранитель».

```
class Editor
{
    protected $content = '';
```

```
    public function type(string $words)
    {
        $this->content = $this->content . ' ' . $words;
    }

    public function getContent()
    {
        return $this->content;
    }

    public function save()
    {
        return new EditorMemento($this->content);
    }

    public function restore(EditorMemento $memento)
    {
        $this->content = $memento->getContent();
    }
}
```

Использование:

```
$editor = new Editor();

// Пишем что-нибудь
$editor->type('This is the first sentence.');
$editor->type('This is second.');

// Сохранение состояния в: This is the first sentence. This is
second.
$saved = $editor->save();

// Пишем ещё
$editor->type('And this is third.');

// Output: Содержимое до сохранения
echo $editor->getContent(); // This is the first sentence. This is
second. And this is third.

// Восстанавливаем последнее сохранённое состояние
$editor->restore($saved);

$editor->getContent(); // This is the first sentence. This is
second.
```

😎 Наблюдатель

Аналогия

Хороший пример: люди, ищущие работу, подписываются на публикации на сайтах вакансий и получают уведомления, когда появляются вакансии, подходящие по параметрам.

Вкратце

Шаблон определяет зависимость между объектами, чтобы при изменении состояния одного из них его «подчинённые» узнавали об этом.

Википедия

> В шаблоне «Наблюдатель» есть объект («субъект»), ведущий список своих «подчинённых» («наблюдателей») и автоматически уведомляющий их о любом изменении своего состояния, обычно с помощью вызова одного из их методов.

Пример

Сначала реализуем людей, ищущих работу, которых нужно уведомлять о появлении вакансий.

```
class JobPost
{
    protected $title;

    public function __construct(string $title)
```

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

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

class JobSeeker implements Observer
{
    protected $name;

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

    public function onJobPosted(JobPost $job)
    {
        // Do something with the job posting
        echo 'Hi ' . $this->name . '! New job posted: '. $job-
>getTitle();
    }
}
```

Теперь реализуем публикации вакансий, на которые люди будут подписываться.

```
class JobPostings implements Observable
{
    protected $observers = [];

    protected function notify(JobPost $jobPosting)
    {
        foreach ($this->observers as $observer) {
            $observer->onJobPosted($jobPosting);
        }
    }

    public function attach(Observer $observer)
    {
        $this->observers[] = $observer;
```

```
    }

    public function addJob(JobPost $jobPosting)
    {
        $this->notify($jobPosting);
    }
}
```

Использование:

```
// Создаём подписчиков
$johnDoe = new JobSeeker('John Doe');
$janeDoe = new JobSeeker('Jane Doe');

// Создаём публикатора и прикрепляем подписчиков
$jobPostings = new JobPostings();
$jobPostings->attach($johnDoe);
$jobPostings->attach($janeDoe);

// Добавляем новую вакансию и смотрим, будут ли уведомлены
подписчики
$jobPostings->addJob(new JobPost('Software Engineer'));

// Output
// Hi John Doe! New job posted: Software Engineer
// Hi Jane Doe! New job posted: Software Engineer
```

 Посетитель

Аналогия

Туристы собрались в Дубай. Сначала им нужен способ попасть туда (виза). После прибытия они будут посещать любую часть города, не спрашивая разрешения, ходить где вздумается. Просто скажите им о каком-нибудь месте — и туристы могут там побывать. Шаблон «Посетитель» помогает добавлять места для посещения.

Вкратце

Шаблон «Посетитель» позволяет добавлять будущие операции для объектов без их модифицирования.

Википедия

> Шаблон «Посетитель» — это способ отделения алгоритма от структуры объекта, в которой он оперирует. Результат отделения — возможность добавлять новые операции в существующие структуры объектов без их модифицирования. Это один из способов соблюдения принципа открытости/закрытости (open/closed principle).

Пример

Возьмём зоопарк: у нас есть несколько видов животных, и нам нужно послушать издаваемые ими звуки.

```
// Место посещения
interface Animal
{
    public function accept(AnimalOperation $operation);
}

// Посетитель
interface AnimalOperation
{
    public function visitMonkey(Monkey $monkey);
    public function visitLion(Lion $lion);
    public function visitDolphin(Dolphin $dolphin);
}
```

Реализуем животных:

```
class Monkey implements Animal
{
    public function shout()
    {
        echo 'Ooh oo aa aa!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitMonkey($this);
    }
}

class Lion implements Animal
{
    public function roar()
    {
        echo 'Roaaar!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitLion($this);
    }
}

class Dolphin implements Animal
{
    public function speak()
    {
        echo 'Tuut tuttu tuutt!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitDolphin($this);
    }
}
```

Реализуем посетителя:

```
class Speak implements AnimalOperation
{
    public function visitMonkey(Monkey $monkey)
    {
        $monkey->shout();
    }

    public function visitLion(Lion $lion)
    {
        $lion->roar();
    }

    public function visitDolphin(Dolphin $dolphin)
    {
        $dolphin->speak();
    }
}
```

Использование:

```
$monkey = new Monkey();
$lion = new Lion();
$dolphin = new Dolphin();

$speak = new Speak();

$monkey->accept($speak);    // Уа-уа-уааааа!
$lion->accept($speak);      // Рррррррррр!
$dolphin->accept($speak);   // Туут тутт туутт!
```

Это можно было сделать просто с помощью иерархии наследования, но тогда пришлось бы модифицировать животных при каждом добавлении к ним новых действий. А здесь менять их не нужно. Например, мы можем добавить животным прыжки, просто создав нового посетителя:

```
class Jump implements AnimalOperation
{
```

```
    public function visitMonkey(Monkey $monkey)
    {
        echo 'Jumped 20 feet high! on to the tree!';
    }

    public function visitLion(Lion $lion)
    {
        echo 'Jumped 7 feet! Back on the ground!';
    }

    public function visitDolphin(Dolphin $dolphin)
    {
        echo 'Walked on water a little and disappeared';
    }
}
```

Использование:

```
$jump = new Jump();

$monkey->accept($speak);   // Ooh oo aa aa!
$monkey->accept($jump);    // Jumped 20 feet high! on to the tree!

$lion->accept($speak);     // Roaaar!
$lion->accept($jump);      // Jumped 7 feet! Back on the ground!

$dolphin->accept($speak);  // Tuut tutt tuutt!
$dolphin->accept($jump);   // Walked on water a little and
disappeared
```

💡 Стратегия

Аналогия

Возьмём пример с пузырьковой сортировкой. Мы её реализовали, но с ростом объёмов данных сортировка стала выполняться очень медленно. Тогда мы сделали быструю сортировку (Quick sort). Алгоритм работает быстрее на больших объёмах, но на маленьких

он очень медленный. Тогда мы реализовали стратегию, при которой для маленьких объёмов данных используется пузырьковая сортировка, а для больших — быстрая.

Вкратце

Шаблон «Стратегия» позволяет переключаться между алгоритмами или стратегиями в зависимости от ситуации.

Википедия

> Шаблон «Стратегия» позволяет при выполнении выбирать поведение алгоритма.

Пример

Возьмём вышеописанный пример. Сначала сделаем интерфейс стратегии и реализации самих стратегий.

```
interface SortStrategy
{
    public function sort(array $dataset): array;
}

class BubbleSortStrategy implements SortStrategy
{
    public function sort(array $dataset): array
    {
        echo "Sorting using bubble sort";

        // Do sorting
        return $dataset;
    }
}

class QuickSortStrategy implements SortStrategy
{
```

```
    public function sort(array $dataset): array
    {
        echo "Sorting using quick sort";

        // Do sorting
        return $dataset;
    }
}
```

Теперь реализуем клиента, который будет использовать нашу стратегию.

```
class Sorter
{
    protected $sorter;

    public function __construct(SortStrategy $sorter)
    {
        $this->sorter = $sorter;
    }

    public function sort(array $dataset): array
    {
        return $this->sorter->sort($dataset);
    }
}
```

Использование:

```
$dataset = [1, 5, 4, 3, 2, 8];

$sorter = new Sorter(new BubbleSortStrategy());
$sorter->sort($dataset); // Output : Пузырьковая сортировка

$sorter = new Sorter(new QuickSortStrategy());
$sorter->sort($dataset); // Output : Быстрая сортировка
```

Состояние

Аналогия

Допустим, в графическом редакторе вы выбрали инструмент «Кисть». Она меняет своё поведение в зависимости от настройки цвета: т. е. рисует линию выбранного цвета.

Вкратце

Шаблон позволяет менять поведение класса при изменении состояния.

Википедия

> Шаблон «Состояние» реализует машину состояний объектно ориентированным способом. Это достигается с помощью:
> - реализации каждого состояния в виде производного класса интерфейса шаблона «Состояние»,
> - реализации переходов состояний (state transitions) посредством вызова методов, определённых вышестоящим классом (superclass).
>
> Шаблон «Состояние» — это в некотором плане шаблон «Стратегия», при котором возможно переключение текущей стратегии с помощью вызова методов, определённых в интерфейсе шаблона.

Пример

Текстовый редактор меняет состояние текста, который вы печатаете, т. е. если выбрано полужирное начертание — то редактор печатает полужирным и т. д.

Сначала сделаем интерфейс состояний и сами состояния:

```
interface WritingState
{
    public function write(string $words);
}

class UpperCase implements WritingState
{
    public function write(string $words)
    {
        echo strtoupper($words);
    }
}

class LowerCase implements WritingState
{
    public function write(string $words)
    {
        echo strtolower($words);
    }
}

class Default implements WritingState
{
    public function write(string $words)
    {
        echo $words;
    }
}
```

Сделаем редактор:

```
class TextEditor
{
    protected $state;
```

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

    public function setState(WritingState $state)
    {
        $this->state = $state;
    }

    public function type(string $words)
    {
        $this->state->write($words);
    }
}
```

Использование:

```
$editor = new TextEditor(new Default());

$editor->type('First line');

$editor->setState(new UpperCase());

$editor->type('Second line');
$editor->type('Third line');

$editor->setState(new LowerCase());

$editor->type('Fourth line');
$editor->type('Fifth line');

// Output:
// First line
// SECOND LINE
// THIRD LINE
// fourth line
// fifth line
```

📒 Шаблонный метод

Аналогия

Допустим, вы собрались строить дома. Этапы будут такими:

- Подготовка фундамента.
- Возведение стен.
- Настил крыши.
- Настил перекрытий.

Порядок этапов никогда не меняется. Вы не настелите крышу до возведения стен — и т. д. Но каждый этап модифицируется: стены, например, можно возвести из дерева, кирпича или газобетона.

Вкратце

«Шаблонный метод» определяет каркас выполнения определённого алгоритма, но реализацию самих этапов делегирует дочерним классам.

Википедия

> «Шаблонный метод» — это поведенческий шаблон, определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

Пример

Допустим, у нас есть программный инструмент, позволяющий тестировать, проводить контроль качества кода (lint), выполнять

сборку, генерировать отчёты сборки (отчёты о покрытии кода, о качестве кода и т. д.), а также развёртывать приложение на тестовом сервере.

Сначала наш базовый класс определяет каркас алгоритма сборки.

```
abstract class Builder
{

    // Шаблонный метод
    final public function build()
    {
        $this->test();
        $this->lint();
        $this->assemble();
        $this->deploy();
    }

    abstract public function test();
    abstract public function lint();
    abstract public function assemble();
    abstract public function deploy();
}
```

Теперь создаём реализации:

```
class AndroidBuilder extends Builder
{
    public function test()
    {
        echo 'Running android tests';
    }

    public function lint()
    {
        echo 'Linting the android code';
    }

    public function assemble()
    {
```

```
        echo 'Assembling the android build';
    }

    public function deploy()
    {
        echo 'Deploying android build to server';
    }
}

class IosBuilder extends Builder
{
    public function test()
    {
        echo 'Running ios tests';
    }

    public function lint()
    {
        echo 'Linting the ios code';
    }

    public function assemble()
    {
        echo 'Assembling the ios build';
    }

    public function deploy()
    {
        echo 'Deploying ios build to server';
    }
}
```

Использование:

```
$androidBuilder = new AndroidBuilder();
$androidBuilder->build();

// Output:
// Выполнение Android-тестов
// Линтинг Android-кода
// Создание Android-сборки
// Развёртывание Android-сборки на сервере

$iosBuilder = new IosBuilder();
$iosBuilder->build();
```

```
// Output:
// Выполнение iOS-тестов
// Линтинг iOS-кода
// Создание iOS-сборки
// Развёртывание iOS-сборки на сервере
```

🚦 Закругляемся

На этом обзор закончен. Я продолжу его улучшать и планирую написать главы про архитектурные шаблоны. Так что периодически заглядывайте.

Похожие публикации

Потоки выполнения и PHP

ПЕРЕВОД

6

AloneCoder • 25 мая 2017 в 19:22

Производительность I/O бэкэнда: Node vs. PHP vs. Java vs. Go

160

ПЕРЕВОД

AloneCoder • 23 мая 2017 в 15:49

Инкремент в PHP

27

ПЕРЕВОД

AloneCoder • 20 июля 2016 в 18:06

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

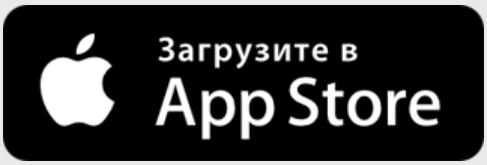
Стивен Хокинг и его работа — что дал ученый человечеству? 8

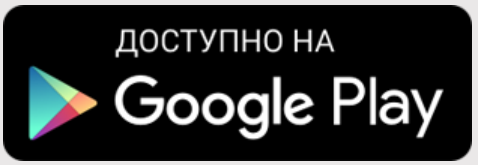
marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия