- Новости
- События и курсы
- Вакансии
- Тесты
- Задачки
- Отвечают эксперты
- Реклама
- Выпустить свой материал
-
- Лучшее за неделю
- Лучшее за месяц
- Лучшее за всё время
-
- **Популярные темы**

Troger

Введите запрос и нажм

- Начинающим
- Алгоритмы
- Планы обучения
- Собеседования
- Web
- JS
- Python
- C++
- Java
- **Все темы**

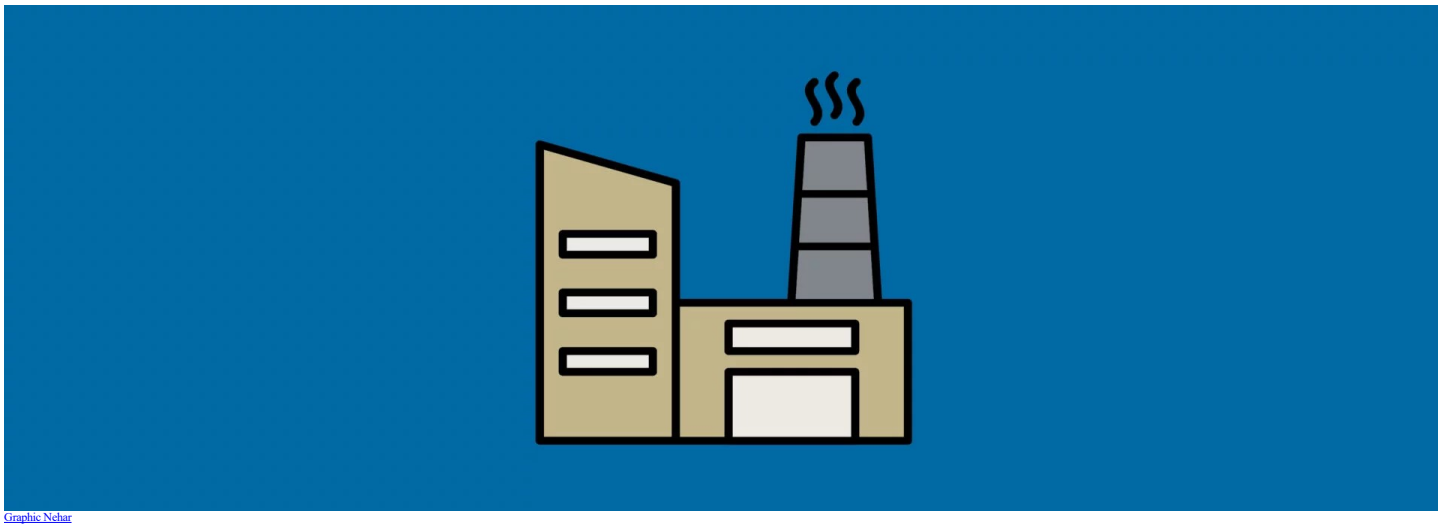
...

. Показать лучшие за неделю Свежие

Го в «Дзен», мы создали)))0)

Шаблоны проектирования простым языком. Часть первая. Порождающие шаблоны

- 1 июня 2017 в 12:14, Переводы
-
- 51 185



Graphic Nehar

Рассказывает Камран Ахмед

Шаблоны проектирования — это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками, как решать определенные проблемы в определенных ситуациях.

Википедия описывает их следующим образом:

> **Шаблон проектирования,** или **паттерн**, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

Будьте осторожны

- шаблоны проектирования не являются решением всех ваших проблем;
- не пытайтесь использовать их в обязательном порядке — это может привести к негативным последствиям. Шаблоны — это подходы к решению проблем, а не решения для поиска проблем;
- если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

Также заметьте, что примеры ниже написаны на PHP 7. Но это не должно вас останавливать, ведь принципы остаются такими же.

Типы шаблонов

Шаблоны бывают следующих трех видов:

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

Если говорить простыми словами, то это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов.

Википедия гласит:

> **Порождающие шаблоны** — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Существуют следующие порождающие шаблоны:

- простая фабрика (Simple Factory);
- фабричный метод (Factory Method);
- абстрактная фабрика (Abstract Factory);
- строитель (Builder);
- прототип (Prototype);
- одиночка (Singleton).

Простая фабрика (Simple Factory)

Википедия гласит:

> В объектно-ориентированном программировании (ООП), **фабрика** — это объект для создания других объектов. Формально фабрика — это функция или метод, который возвращает объекты изменяющегося прототипа или класса из некоторого вызова метода, который считается «новым».

Пример из жизни: Представьте, что вам надо построить дом, и вам нужны двери. Было бы глупо каждый раз, когда вам нужны двери, надевать вашу столярную форму и начинать делать дверь. Вместо этого вы делаете её на фабрике.

Простыми словами: Простая фабрика генерирует экземпляр для клиента, не раскрывая никакой логики.

Перейдем к коду. У нас есть интерфейс `Door` 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; } }
```

Затем у нас есть наша `DoorFactory`, которая делает дверь и возвращает её:

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

И затем мы можем использовать всё это:

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

Когда использовать: Когда создание объекта — это не просто несколько присвоений, а какая-то логика, тогда имеет смысл создать отдельную фабрику вместо повторения одного и того же кода повсюду.

Пример на Java.

Фабричный метод (Fabric Method)

Википедия гласит:

> **Фабричный метод** — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Пример из жизни: Рассмотрим пример с менеджером по найму. Невозможно одному человеку провести собеседования со всеми кандидатами на все вакансии. В зависимости от вакансии он должен распределить этапы собеседования между разными людьми.

Простыми словами: Менеджер предоставляет способ делегирования логики создания экземпляра дочерним классам.

Перейдём к коду. Рассмотрим приведенный выше пример про HR-менеджера. Изначально у нас есть интерфейс `Interviewer` и несколько реализаций для него:

```
interface Interviewer { public function askQuestions(); } class Developer implements Interviewer { public function askQuestions() { echo 'Спрашивает про шаблоны проектирования!'; } } class CommunityExecutive implements Interviewer { public function askQuestions() { echo 'Спрашивает о работе с сообществом'; } }
```

Теперь создадим нашего `HiringManager`:

```
abstract class HiringManager { // Фабричный метод abstract public function makeInterviewer(): Interviewer; public function takeInterview() { $interviewer = $this->makeInterviewer(); $interviewer->askQuestions(); } }
```

И теперь любой дочерний класс может расширять его и предоставлять необходимого интервьюера:

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

Пример использования:

```
$devManager = new DevelopmentManager(); $devManager->takeInterview(); // Вывод: Спрашивает о шаблонах проектирования! $marketingManager = new MarketingManager(); $marketingManager->takeInterview(); // Вывод: Спрашивает о работе с сообществом
```

Когда использовать: Полезен, когда есть некоторая общая обработка в классе, но необходимый подкласс динамически определяется во время выполнения. Иными словами, когда клиент не знает, какой именно подкласс ему может понадобиться.

Примеры на Java и Python.

Абстрактная фабрика (Abstract Factory)

Википедия гласит:

> **Абстрактная фабрика** — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Пример из жизни: Расширим наш пример про двери из простой фабрики. В зависимости от ваших нужд вам понадобится деревянная дверь из одного магазина, железная дверь — из другого или пластиковая — из третьего. Кроме того, вам понадобится соответствующий специалист: столяр для деревянной двери, сварщик для железной двери и так далее. Как вы можете заметить, тут есть зависимость между дверьми.

Простыми словами: Фабрика фабрик. Фабрика, которая группирует индивидуальные, но связанные/зависимые фабрики без указания их конкретных классов.

Обратимся к коду. Используем пример про двери. Сначала у нас есть интерфейс `Door` и несколько его реализаций:

```
interface Door { public function getDescription(); } class WoodenDoor implements Door { public function getDescription() { echo 'Я деревянная дверь'; } } class IronDoor implements Door { public function getDescription() { echo 'Я железная дверь'; } }
```

Затем у нас есть несколько `DoorFittingExpert` для каждого типа дверей:

```
interface DoorFittingExpert { public function getDescription(); } class Welder implements DoorFittingExpert { public function getDescription() { echo 'Я работаю только с железными дверьми'; } } class Carpenter implements DoorFittingExpert { public function getDescription() { echo 'Я работаю только с деревянными дверьми'; } }
```

Теперь у нас есть `DoorFactory`, которая позволит нам создать семейство связанных объектов. То есть фабрика деревянных дверей предоставит нам деревянную дверь и эксперта по деревянным дверям. Аналогично для железных дверей:

```
interface DoorFactory { public function makeDoor(): Door; public function makeFittingExpert(): DoorFittingExpert; } // Деревянная фабрика вернет деревянную дверь и столяра class WoodenDoorFactory implements DoorFactory { public function makeDoor(): Door { return new WoodenDoor(); } public function makeFittingExpert(): DoorFittingExpert { return new Carpenter(); } } // Железная фабрика вернет железную дверь и сварщика class IronDoorFactory implements DoorFactory { public function makeDoor(): Door { return new IronDoor(); } public function makeFittingExpert(): DoorFittingExpert { return new Welder(); } }
```

Пример использования:

```
$woodenFactory = new WoodenDoorFactory(); $door = $woodenFactory->makeDoor(); $expert = $woodenFactory->makeFittingExpert(); $door->getDescription(); // Вывод: Я деревянная дверь $expert->getDescription(); // Вывод: Я работаю только с деревянными дверьми // Аналогично для железной двери $ironFactory = new IronDoorFactory(); $door = $ironFactory->makeDoor(); $expert = $ironFactory->makeFittingExpert(); $door->getDescription(); // Вывод: Я железная дверь $expert->getDescription(); // Вывод: Я работаю только с железными дверьми
```

Как вы можете заметить, фабрика деревянных дверей инкапсулирует столяра и деревянную дверь, а фабрика железных дверей инкапсулирует железную дверь и сварщика. Это позволило нам убедиться, что для каждой двери мы получим нужного нам эксперта.

Когда использовать: Когда есть взаимосвязанные зависимости с не очень простой логикой создания.

Примеры на Java и Python.

Строитель (Builder)

Википедия гласит:

> **Строитель** — порождающий шаблон проектирования, который предоставляет способ создания составного объекта. Предназначен для решения проблемы антипаттерна «Телескопический конструктор».

Пример из жизни: Представьте, что вы пришли в McDonalds и заказали конкретный продукт, например, БигМак, и вам готовят его без лишних вопросов. Это пример простой фабрики. Но есть случаи, когда логика создания может включать в себя больше шагов. Например, вы хотите индивидуальный сэндвич в Subway: у вас есть несколько вариантов того, как он будет сделан. Какой хлеб вы хотите? Какие соусы использовать? Какой сыр? В таких случаях на помощь приходит шаблон «Строитель».

Простыми словами: Шаблон позволяет вам создавать различные виды объекта, избегая засорения конструктора. Он полезен, когда может быть несколько видов объекта или когда необходимо множество шагов, связанных с его созданием.

Давайте я покажу на примере, что такое «Телескопический конструктор». Когда-то мы все видели конструктор вроде такого:

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

Как вы можете заметить, количество параметров конструктора может резко увеличиться, и станет сложно понимать расположение параметров. Кроме того, этот список параметров будет продолжать расти, если вы захотите добавить новые варианты. Это и есть «Телескопический конструктор».

Перейдем к примеру в коде. Адекватной альтернативой будет использование шаблона «Строитель». Сначала у нас есть `Burger`, который мы хотим создать:

```
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; } }
```

Затем мы берём «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); } }
```

Пример использования:

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

Когда использовать: Когда может быть несколько видов объекта и надо избежать «телескопического конструктора». Главное отличие от «фабрики» — это то, что она используется, когда создание занимает один шаг, а «строитель» применяется при множестве шагов.

Примеры на Java и Python.

Прототип (Prototype)

Википедия гласит:

> Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Он позволяет уйти от реализации и позволяет следовать принципу «программирование через интерфейсы». В качестве возвращающего типа указывается интерфейс / абстрактный класс на вершине иерархии, а классы-наследники могут подставить туда наследника, реализующего этот тип.

Пример из жизни: Помните Долли? Овечка, которая была клонирована. Не будем углубляться, главное — это то, что здесь все вращается вокруг клонирования.

Простыми словами: Прототип создает объект, основанный на существующем объекте при помощи клонирования.

То есть он позволяет вам создавать копию существующего объекта и модернизировать его согласно вашим нуждам, вместо того, чтобы создавать объект заново.

Обратимся к коду. В PHP это может быть легко реализовано с использованием `clone`:

```
class Sheep { protected $name; protected $category; public function __construct(string $name, string $category = 'Горная овечка') { $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 он может быть клонирован следующим образом:

```
$original = new Sheep('Джолли'); echo $original->getName(); // Джолли echo $original->getCategory(); // Горная овечка // Клонируем и модифицируем то что нужно $cloned = clone $original; $cloned->setName('Долли'); echo $cloned->getName(); // Долли echo $cloned->getCategory(); // Горная овечка
```

Также вы можете использовать волшебный метод `__clone` для изменения клонирующего поведения.

Когда использовать: Когда необходим объект, похожий на существующий объект, либо когда создание будет дороже клонирования.

Примеры на Java и Python.

Одиночка (Singleton)

Википедия гласит:

> **Одиночка** — порождающий шаблон проектирования, гарантирующий, что в однопроцессном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

Пример из жизни: В стране одновременно может быть только один президент. Один и тот же президент должен действовать, когда того требуют обстоятельства. Президент здесь является одиночкой.

Простыми словами: Обеспечивает тот факт, что создаваемый объект является единственным объектом своего класса.

Вообще шаблон одиночка признан антипаттерном, необходимо избегать его чрезмерного использования. Он необязательно плох и может иметь полезные применения, но использовать его надо с осторожностью, потому что он вводит глобальное состояние в ваше приложение и его изменение в одном месте может повлиять на другие части приложения, что вызовет трудности при отладке. Другой минус — это то, что он делает ваш код связанным.

Прим. перев. Подробнее о подводных камнях шаблона одиночка читайте в нашей статье.

Перейдем к коду. Чтобы создать одиночку, сделайте конструктор приватным, отключите клонирование и расширение 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 } }
```

Пример использования:

```
$president1 = President::getInstance(); $president2 = President::getInstance(); var_dump($president1 === $president2); // true
```

Пример на Java.

Перевод статьи «Design Patterns for Humans»

- PHP, Для продолжающих, Паттерны проектирования, Шаблоны проектирования простым языком

Рассылка «Аргументы и функции»

Только самые важные IT-новости

Введите свой e-mail Подписаться

События и курсы

19 мая, Москва: хакатон DIGITAX

20 января — 1 июня, онлайн: международная олимпиада «IT-Планета»

23–24 апреля, Москва: конференция FinTech Day 2019

23 апреля, Москва: конференция «Роботизация бизнес-процессов 2019»

25 апреля, Москва: форум Open Agile Day

25 апреля, Москва: %%title%%

26–27 апреля, Санкт-Петербург: конференция HR API 2019

Все события и курсы

Вакансии

Системный аналитик DWH/BI
Москва

QA automation engineer/QA автоматизатор тестирования
Москва, до 150 000 ₽

Backend-разработчик
Санкт-Петербург, от 170 000 до 220 000 ₽

Senior Java developer
Москва, от 200 000 до 300 000 ₽

Big Data инженер
Москва

Все вакансии