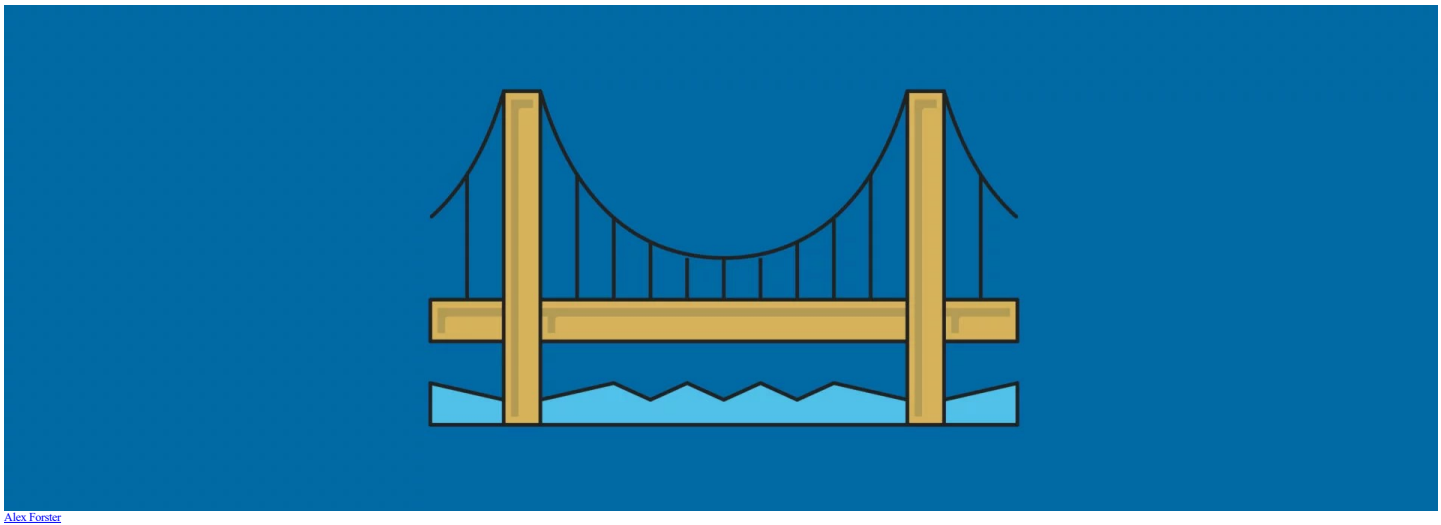
Шаблоны проектирования простым языком. Часть вторая. Структурные шаблоны

- 4 июля 2017 в 14:34, Переводы
-
- 21 267



Alex Forster

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 решения определенных проблем 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 — о них мы рассказываем в этой статье.
3. Поведенческие.

Простыми словами: Структурные шаблоны в основном связаны с композицией объектов, другими словами, с тем, как сущности могут использовать друг друга. Ещё одним объяснением было бы то, что они помогают ответить на вопрос «Как создать программный компонент?».

Википедия гласит:

> **Структурные шаблоны** — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Список структурных шаблонов проектирования:

- адаптер (Adapter);
- мост (Bridge);
- компоновщик (Composite);
- декоратор (Decorator);
- фасад (Facade);
- приспособленец (Flyweight);
- заместитель (Proxy).

Адаптер (Adapter)

Википедия гласит:

> **Адаптер** — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Пример из жизни: Представим, что у вас на карте памяти есть какие-то изображения и вам надо перенести их на ваш компьютер. Чтобы это сделать, вам нужен какой-то адаптер, который совместим с портами вашего компьютера. В этом случае карт-ридер — это адаптер. Другим примером будет блок питания. Вилку с тремя ножками нельзя вставить в розетку с двумя отверстиями. Для того, чтобы она подошла, надо использовать адаптер. Ещё одним примером будет переводчик, переводящий слова одного человека для другого.

Простыми словами: Шаблон позволяет обернуть несовместимые объекты в адаптер, чтобы сделать их совместимыми с другим классом.

Обратимся к коду. Представим игру, в которой охотник охотится на львов.

Изначально у нас есть интерфейс `Lion`, который реализует всех львов:

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

И `Hunter` охотится на любую реализацию интерфейса `Lion`:

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

Теперь представим, что нам надо добавить `WildDog` в нашу игру, на которую наш `Hunter` также мог бы охотиться. Но мы не можем сделать это напрямую, потому что у `WildDog` другой интерфейс. Чтобы сделать её совместимой с нашим `Hunter`, нам надо создать адаптер:

```
// Это надо добавить в игру class WildDog { public function bark() { } } // Адаптер, чтобы сделать WildDog совместимой с нашей игрой class WildDogAdapter implements Lion { protected $dog; public function __construct(WildDog $dog) { $this->dog = $dog; } public function roar() { $this->dog->bark(); } }
```

Способ применения:

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

Примеры на Java и Python.

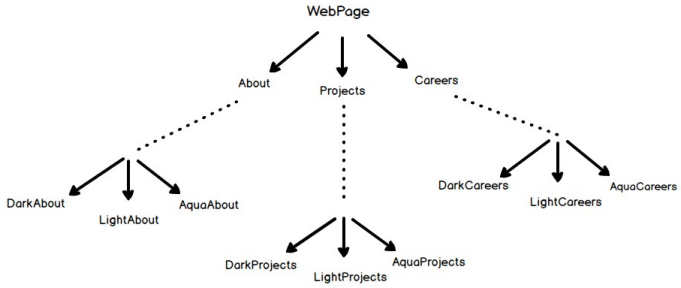
Мост (Bridge)

Википедия гласит:

> **Мост** — структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Шаблон мост использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

Пример из жизни: Представим, что у вас есть сайт с разными страницами, и вам надо разрешить пользователям менять их тему. Что вы будете делать? Создавать множественные копии каждой страницы для каждой темы или просто отдельную тему, которую пользователь сможет выбрать сам? Шаблон мост позволяет вам сделать второе.

Without Bridge



With Bridge

Простыми словами: Шаблон мост — это предпочтение композиции над наследованием. Детали реализации передаются из одной иерархии в другой объект с отдельной иерархией.

Обратимся к примеру в коде. Возьмем пример с нашими страницами. У нас есть иерархия `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 "Страница с информацией в " . $this->theme->getColor(); } } class Careers implements WebPage { protected $theme; public function __construct(Theme $theme) { $this->theme = $theme; } public function getContent() { return "Страница карьеры в " . $this->theme->getColor(); } }
```

И отдельная иерархия `Theme`:

```
interface Theme { public function getColor(); } class DarkTheme implements Theme { public function getColor() { return 'темной теме'; } } class LightTheme implements Theme { public function getColor() { return 'светлой теме'; } } class AquaTheme implements Theme { public function getColor() { return 'голубой теме'; } }
```

Применение в коде:

```
$darkTheme = new DarkTheme(); $about = new About($darkTheme); $careers = new Careers($darkTheme); echo $about->getContent(); // "Страница информации в темной теме"; echo $careers->getContent(); // "Страница карьеры в темной теме";
```

Примеры на Java и Python.

Компоновщик (Composite)

Википедия гласит:

> **Компоновщик** — структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

Пример из жизни: Каждая организация скомпонована из сотрудников. У каждого сотрудника есть одинаковые свойства, такие как зарплата, обязанности, отчётность и т.д.

Простыми словами: Шаблон компоновщик позволяет клиентам работать с индивидуальными объектами в едином стиле.

Обратимся к коду. Возьмем наш пример с рабочими. У нас есть `Employee` разных типов:

```
interface Assignee { public function canHandleTask($task): bool; public function takeTask($task); } class Employee implements Assignee { // реализуем методы интерфейса } class Team implements Assignee { /** @var Assignee[] */ private $assignees; // вспомогательные методы для управления композитом: public function add($assignee); public function remove($assignee); // метода интерфейса Employee public function canHandleTask($task): bool { foreach ($this->assignees as $assignee) if ($assignee->canHandleTask($task)) return true; return false; } public function takeTask($task) { // может быть разная имплементация - допустим, некоторые задания требуют нескольких человек из команды одновременно // в простейшем случае берем первого незанятого работника среди this->assignees $assignee = ...; $assignee->takeTask($task); } }
```

Теперь у нас есть `TaskManager`:

```
class TaskManager { private $assignees; public function performTask($task) { foreach ($this->assignees as $assignee) { if ($assignee->canHandleTask($task)) { $assignee->takeTask($task); return; } } throw new Exception('Cannot handle the task - please hire more people'); } }
```

Способ применения:

```
$employee1 = new Employee(); $employee2 = new Employee(); $employee3 = new Employee(); $employee4 = new Employee(); $team1 = new Team([$employee3, $employee4]); // ВНИМАНИЕ: передаем команду в taskManager как единый композит. // Сам taskManager не знает, что это команда и работает с ней без модификации своей логики. $taskManager = new TaskManager([$employee1, $employee2, $team1]); $taskManager->preformTask($task);
```

Примеры на Java и Python.

Декоратор (Decorator)

Википедия гласит:

> **Декоратор** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Пример из жизни: Представим, что у вас есть свой автосервис. Как вы будете рассчитывать сумму в счете за услуги? Вы выбираете одну услугу и динамически добавляете к ней цены на предоставляемые услуги, пока не получите окончательную стоимость. Здесь каждый тип сервиса является декоратором.

Простыми словами: Шаблон декоратор позволяет вам динамически изменять поведение объекта во время работы, оборачивая их в объект класса декоратора.

Перейдем к коду. Возьмем пример с кофе. Изначально у нас есть простой `Coffee` и реализующий его интерфейс:

```
interface Coffee { public function getCost(); public function getDescription(); } class SimpleCoffee implements Coffee { public function getCost() { return 10; } public function getDescription() { return 'Простой кофе'; } }
```

Мы хотим сделать код расширяемым, чтобы при необходимости можно было изменять его. Давайте сделаем некоторые дополнения (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() . ', молоко'; } } 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() . ', сливки'; } } 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() . ', ваниль'; } }
```

А теперь приготовим `Coffee`:

```
$someCoffee = new SimpleCoffee(); echo $someCoffee->getCost(); // 10 echo $someCoffee->getDescription(); // Простой кофе $someCoffee = new MilkCoffee($someCoffee); echo $someCoffee->getCost(); // 12 echo $someCoffee->getDescription(); // Простой кофе, молоко $someCoffee = new WhipCoffee($someCoffee); echo $someCoffee->getCost(); // 17 echo $someCoffee->getDescription(); // Простой кофе, молоко, сливки $someCoffee = new VanillaCoffee($someCoffee); echo $someCoffee->getCost(); // 20 echo $someCoffee->getDescription(); // Простой кофе, молоко, сливки, ваниль
```

Примеры на Java и Python.

Фасад (Facade)

Википедия гласит:

> **Фасад** — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Пример из жизни: Как вы включаете компьютер? Нажимаю на кнопку включения, скажете вы. Это то, во что вы верите, потому что вы используете простой интерфейс, который компьютер предоставляет для доступа снаружи. Внутри же должно произойти гораздо больше вещей. Этот простой интерфейс для сложной подсистемы называется фасадом.

Простыми словами: Шаблон фасад предоставляет упрощенный интерфейс для сложной системы.

Перейдем к примерам в коде. Возьмем пример с компьютером. Изначально у нас есть класс `Computer`:

```
class Computer { public function getElectricShock() { echo "Ай!"; } public function makeSound() { echo "Бип-бип!"; } public function showLoadingScreen() { echo "Загрузка.."; } public function bam() { echo "Готов к использованию!"; } public function closeEverything() { echo "Буп-буп-буп-бзззз!"; } public function sooth() { echo "Zzzzz"; } public function pullCurrent() { echo "Аах!"; } }
```

Затем у нас есть 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(); } }
```

Пример использования:

```
$computer = new ComputerFacade(new Computer()); $computer->turnOn(); // Ай! Бип-бип! Загрузка.. Готов к использованию! $computer->turnOff(); // Буп-буп-буп-бзззз! Аах! Zzzzz
```

Примеры на Java и Python.

Приспособленец (Flyweight)

Википедия гласит:

> **Приспособленец** — структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

Пример из жизни: Вы когда-нибудь заказывали чай в уличном ларьке? Там зачастуют готовят не одну чашку, которую вы заказали, а гораздо большую емкость. Это делается для того, чтобы экономить ресурсы (газ/электричество). Газ/электричество в этом примере и являются приспособленцами, ресурсы которых делятся (sharing).

Простыми словами: Приспособленец используется для минимизации использования памяти или вычислительной стоимости путем разделения ресурсов с наибольшим количеством похожих объектов.

Перейдем к примерам в коде. Возьмем наш пример с чаем. Изначально у нас есть различные виды `Tea` и `TeaMaker`:

```
// Все, что будет закешировано, является приспособленцем. // Типы чая здесь будут приспособленцами. class KarakTea { } //Ведет себя как фабрика и сохраняет чай class TeaMaker { protected $availableTea = []; public function make($preference) { if (empty($this->availableTea[$preference])) { $this->availableTea[$preference] = new KarakTea(); } return $this->availableTea[$preference]; } }
```

Теперь у нас есть `TeaShop`, который принимает заказы и выполняет их:

```
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; } } }
```

Пример использования:

```
$teaMaker = new TeaMaker(); $shop = new TeaShop($teaMaker); $shop->takeOrder('меньше сахара', 1); $shop->takeOrder('больше молока', 2); $shop->takeOrder('без сахара', 5); $shop->serve(); // Подаем чай на первый стол // Подаем чай на второй стол // Подаем чай на пятый стол
```

Примеры на Java и Python.

Заместитель (Proxy)

Википедия гласит:

> **Заместитель** — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

Пример из жизни: Вы когда-нибудь использовали карту доступа, чтобы пройти через дверь? Есть несколько способов открыть дверь: например, она может быть открыта при помощи карты доступа или нажатия кнопки, которая обходит защиту. Основная функциональность двери — это открытие, но заместитель, добавленный поверх этого, добавляет функциональность. Но лучше я объясню это на примере кода чуть ниже.

Простыми словами: Используя шаблон заместитель, класс отображает функциональность другого класса.

Перейдем к коду. Возьмем наш пример с безопасностью. Сначала у нас есть интерфейс `Door` и его реализация:

```
interface Door { public function open(); public function close(); } class LabDoor implements Door { public function open() { echo "Открытие дверь лаборатории"; } public function close() { echo "Закрытие двери лаборатории"; } }
```

Затем у нас есть заместитель `Security` для защиты любых наших дверей:

```
class Security { protected $door; public function __construct(Door $door) { $this->door = $door; } public function open($password) { if ($this->authenticate($password)) { $this->door->open(); } else { echo "Нет! Это невозможно."; } } public function authenticate($password) { return $password === '$ecr@t'; } public function close() { $this->door->close(); } }
```

Пример использования:

```
$door = new Security(new LabDoor()); $door->open('invalid'); // Нет! Это невозможно. $door->open('$ecr@t'); // Открытие двери лаборатории $door->close(); // Закрытие двери лаборатории
```

Другим примером будет реализация маппинга данных. Например, недавно я создал ODM (Object Data Mapper) для MongoDB, используя этот шаблон, где я написал заместитель вокруг классов mongo и использовал магический метод `__call()`. Все вызовы методов были замещены оригинальным классом mongo, и полученный результат возвращался без изменений, но в случае `find` или `findOne` данные сопоставлялись необходимому классу, и возвращались в объект вместо `Cursor`.

Примеры на Java и Python.

Перевод статьи «Design Patterns for Humans»

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

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