- 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**

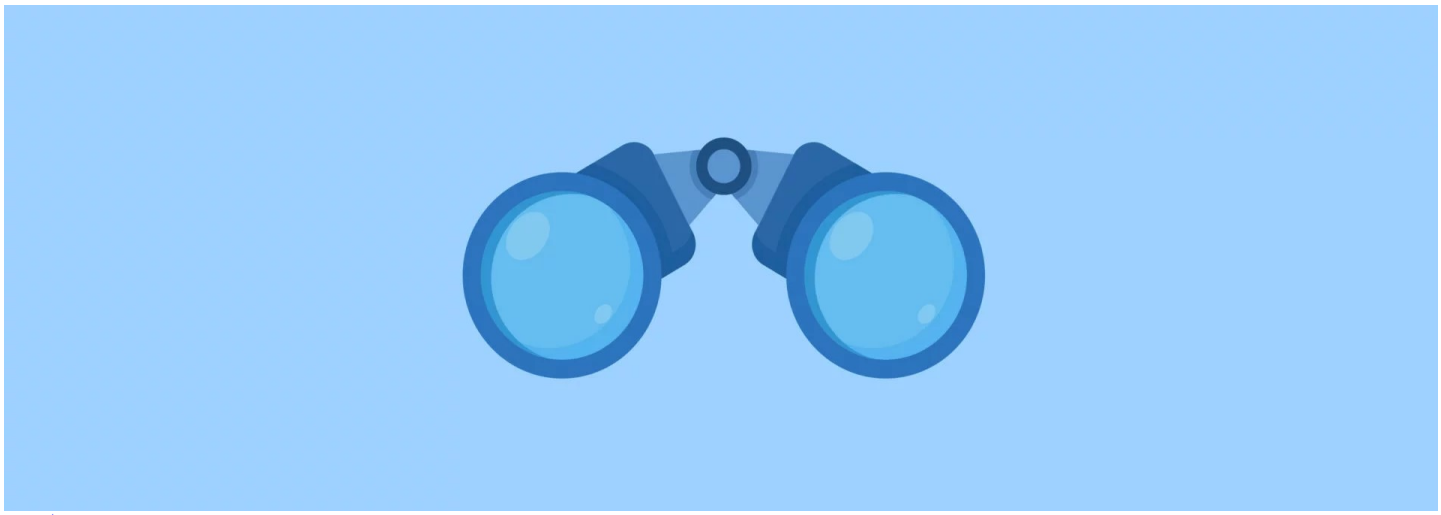
...

. Показать лучшие за неделю Свежие

Го в «Дзен», мы создали )))0)

# Шаблоны проектирования простым языком. Часть третья. Поведенческие шаблоны

- 4 июля 2017 в 14:40, Переводы
- 
- 19 016



popcornarts

*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 насильно использовать их, из-за этого могут произойти плохие вещи. Шаблоны — решения 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 — о них мы рассказываем в этой статье.

**Простыми словами:** Поведенческие шаблоны связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том, что они не просто описывают структуру, но также описывают шаблоны для передачи сообщений / связи между ними. Или, другими словами, они помогают ответить на вопрос «Как запустить поведение в программном компоненте?»

Википедия гласит:

> **Поведенческие шаблоны** — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Поведенческие шаблоны:

- цепочка обязанностей (Chain of Responsibility);
- команда (Command);
- итератор (Iterator);
- посредник (Mediator);
- хранитель (Memento);
- наблюдатель (Observer);
- посетитель (Visitor);
- стратегия (Strategy);
- состояние (State);
- шаблонный метод (Template Method).

## Цепочка обязанностей (Chain of Responsibility)

Википедия гласит:

> **Цепочка обязанностей** — поведенческий шаблон проектирования предназначенный для организации в системе уровней ответственности.

**Пример из жизни:** например, у вас есть три платежных метода (A, B и C), настроенных на вашем банковском счёте. На каждом лежит разное количество денег. На A есть 100 долларов, на B есть 300 долларов и на C — 1000 долларов. Предпочтение отдается в следующем порядке: A, B и C. Вы пытаетесь заказать что-то, что стоит 210 долларов. Используя цепочку обязанностей, первым на возможность оплаты будет проверен метод A, и в случае успеха пройдет оплата и цепь разорвется. Если нет, то запрос перейдет к методу B для аналогичной проверки. Здесь A, B и C — это звенья цепи, а все явление — цепочка обязанностей.

**Простыми словами:** цепочка обязанностей помогает строить цепочки объектов. Запрос входит с одного конца и проходит через каждый объект, пока не найдет подходящий обработчик.

Обратимся к коду. Приведем пример с банковскими счетами. Изначально у нас есть базовый `Account` с логикой для соединения счетов цепью и некоторые 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('Оплата %s, используя %s' . PHP_EOL, $amountToPay, get_called_class()); } elseif ($this->successor) { echo sprintf('Нельзя заплатить, используя %s. Обработка ..' . PHP_EOL, get_called_class()); $this->successor->pay($amountToPay); } else { throw new Exception('Ни на одном из аккаунтов нет необходимого количества денег'); } } 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 приготовим цепь, используя объявленные выше звенья (например, Bank, Paypal, Bitcoin):

```
// Подготовим цепь // $bank->$paypal->$bitcoin // // Первый по приоритету банк // Если нельзя через банк, то Paypal // Если нельзя через Paypal, то Bitcoin $bank = new Bank(100); // Банк с балансом 100 $paypal = new Paypal(200); // Paypal с балансом 200 $bitcoin = new Bitcoin(300); // Bitcoin с балансом 300 $bank->setNext($paypal); $paypal->setNext($bitcoin); // Попробуем оплатить через банк $bank->pay(259); // Вывод // ============== // Нельзя заплатить, используя Bank. Обработка .. // Нельзя заплатить, используя Paypal. Обработка ..: // Оплата 259, используя Bitcoin!
```

Примеры на Java и Python.

## Команда (Command)

Википедия гласит:

> **Команда** — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

**Пример из жизни:** Типичный пример: вы заказываете еду в ресторане. Вы (т.е. `Client`) просите официанта (например, `Invoker`) принести еду (то есть `Command`), а официант просто перепрaвляет запрос шеф-повару (то есть `Receiver`), который знает, что и как готовить. Другим примером может быть то, что вы (`Client`) включаете (`Command`) телевизор (`Receiver`) с помощью пульта дистанционного управления (`Invoker`).

**Простыми словами:** Позволяет вам инкапсулировать действия в объекты. Основная идея, стоящая за шаблоном — это предоставление средств, для разделения клиента и получателя.

Обратимся к коду. Изначально у нас есть получатель `Bulb`, в котором есть реализация каждого действия, которое может быть выполнено:

```
// Получатель class Bulb { public function turnOn() { echo "Лампочка загорелась"; } public function turnOff() { echo "Темнота!"; } }
```

Затем у нас есть интерфейс `Command`, которая каждая команда должна реализовывать, и затем у нас будет набор команд:

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

Затем у нас есть `Invoker`, с которым клиент будет взаимодействовать для обработки любых команд:

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

Наконец, мы можем увидеть, как использовать нашего клиента:

```
$bulb = new Bulb(); $turnOn = new TurnOn($bulb); $turnOff = new TurnOff($bulb); $remote = new RemoteControl(); $remote->submit($turnOn); // Лампочка загорелась! $remote->submit($turnOff); // Темнота!
```

Шаблон команда может быть использован для реализации системы, основанной на транзакциях, где вы сохраняете историю команд, как только их выполняете. Если окончательная команда успешно выполнена, то все хорошо, иначе алгоритм просто перебирает историю и продолжает выполнять отмену для всех выполненных команд.

Примеры на Java и Python.

## Итератор (Iterator)

Википедия гласит:

> **Итератор** — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

**Пример из жизни:** Старый радионабор будет хорошим предметом итератора, где пользователь может начать искать сигнал на каком-то канале и затем использовать кнопки переключения на следующий и предыдущий канал для перехода между соответствующими каналами. Или используем пример телевизора, где вы можете нажимать кнопки следующего или предыдущего канала для перехода через последовательные каналы, или, иными словами, они предоставляют интерфейс для итерирования между соответствующими каналами, песнями или радиостанциями.

**Простыми словами:** Представляет способ доступа к элементам объекта без показа базового представления.

Обратимся к примерам в коде. В PHP очень просто реализовать это, используя SPL (Standard PHP Library). Приводя наш пример с радиостанциями, изначально у нас есть `Radiostation`:

```
class RadioStation { protected $frequency; public function __construct(float $frequency) { $this->frequency = $frequency; } public function getFrequency(): float { return $this->frequency; } }
```

Затем у нас есть 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]); } }
```

Пример использования:

```
$stationList = new StationList(); // Добавление станций $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)); // Удалит 89 станцию
```

Примеры на Java и Python.

## Посредник (Mediator)

Википедия гласит:

> **Посредник** — поведенческий шаблон проектирования, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность, и избавляя объекты, от необходимости явно ссылаться друг на друга.

**Пример из жизни:** Общим примером будет, 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.

**Простыми словами:** 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.

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

Изначально у нас есть посредник `ChatRoomMediator`:

```
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; } }
```

Затем у нас есть наши `User` (коллеги):

```
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); } }
```

Пример использования:

```
$mediator = new ChatRoom(); $john = new User('John Doe', $mediator); $jane = new User('Jane Doe', $mediator); $john->send('Привет!'); $jane->send('Привет!'); // Вывод // Feb 14, 10:58 [John]: Привет! // Feb 14, 10:58 [Jane]: Привет!
```

Примеры на Java и Python.

## Хранитель (Memento)

Википедия гласит:

> **Хранитель** — поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в этом состоянии.

**Пример из жизни:** 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).

**Простыми словами:** Шаблон хранитель фиксирует и хранит текущее состояние объекта, чтобы оно легко восстанавливалось.

Обратимся к коду. Возьмем наш пример текстового редактора, который время от времени сохраняет состояние, которое вы можете восстановить.

Изначально у нас есть наш объект `EditorMemento`, который может содержать состояние редактора:

```
class EditorMemento { protected $content; public function __construct(string $content) { $this->content = $content; } public function getContent() { return $this->content; } }
```

Затем у нас есть наш `Editor` (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(); } }
```

Пример использования:

```
$editor = new Editor(); // Печатаем что-нибудь $editor->type('Это первое предложение.'); $editor->type('Это второе.'); // Сохраняем состояние для восстановления : Это первое предложение. Это второе. $saved = $editor->save(); // Печатаем ещё $editor->type('И это третье.'); // Вывод: Данные до сохранения echo $editor->getContent(); // Это первое предложение. Это второе. И это третье. // Восстановление последнего сохранения $editor->restore($saved); $editor->getContent(); // Это первое предложение. Это второе.
```

Примеры на Java и Python.

## Наблюдатель (Observer)

Википедия гласит:

> **Наблюдатель** — поведенческий шаблон проектирования, также известен как «подчинённые» (Dependents). Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

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

**Простыми словами:** Шаблон определяет зависимость между объектами, чтобы при изменении состояния одного из них зависимые от него узнавали об этом.

Обратимся к коду. Приводя наш пример. Изначально у нас есть `JobSeeker`, которые ищут работы `JobPost` и должны быть уведомлены о её появлении:

```
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) { // Делаем что-то с публикациями вакансий echo 'Привет ' . $this->name . '! Появилась новая работа: '. $job->getTitle(); } }
```

Затем мы делаем публикации `JobPostings` на которые соискатели могут 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); } }
```

Пример использования:

```
// Создаем соискателей $johnDoe = new JobSeeker('John Doe'); $janeDoe = new JobSeeker('Jane Doe'); // Создаем публикацию и добавляем подписчика $jobPostings = new JobPostings(); $jobPostings->attach($johnDoe); $jobPostings->attach($janeDoe); // Добавляем новую работу и смотрим получит ли соискатель уведомление $jobPostings->addJob(new JobPost('Software Engineer')); // Вывод // Привет John Doe! Появилась новая работа: Software Engineer // Привет Jane Doe! Появилась новая работа: Software Engineer
```

Примеры на Java и Python.

## Посетитель (Visitor)

Википедия гласит:

> **Посетитель** — поведенческий шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов. При изменении visitor нет необходимости изменять обслуживаемые классы.

**Пример из жизни:** 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.

**Простыми словами:** Шаблон посетитель позволяет добавлять будущие операции для объектов без их модифицирования.

Перейдем к примерам в коде. Возьмём зоопарк: у нас есть несколько видов `Animal`, и нам нужно послушать издаваемые ими звуки.

```
// Посещаемый 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); }
```

Затем у нас есть реализация для животных:

```
class Monkey implements Animal { public function shout() { echo 'Y-y-a-a!'; } public function accept(AnimalOperation $operation) { $operation->visitMonkey($this); } } class Lion implements Animal { public function roar() { echo 'pppp!'; } public function accept(AnimalOperation $operation) { $operation->visitLion($this); } } class Dolphin implements Animal { public function speak() { echo '*звуки дельфина*!'; // Я понятия не имею как описать их звуки } public function accept(AnimalOperation $operation) { $operation->visitDolphin($this); } }
```

Давайте 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(); } }
```

Пример использования:

```
$monkey = new Monkey(); $lion = new Lion(); $dolphin = new Dolphin(); $speak = new Speak(); $monkey->accept($speak); // У-у-а-а! $lion->accept($speak); // Pppp! $dolphin->accept($speak); // *звуки дельфина*!
```

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 'Прыгает на 20 футов!'; } public function visitLion(Lion $lion) { echo 'Прыгает на 7 футов!'; } public function visitDolphin(Dolphin $dolphin) { echo 'Появился над водой и исчез!'; } }
```

Пример использования:

```
$jump = new Jump(); $monkey->accept($speak); // У-у-а-а! $monkey->accept($jump); // Прыгает на 20 футов! $lion->accept($speak); // Pppp! $lion->accept($jump); // Прыгает на 7 футов! $dolphin->accept($speak); // *звуки дельфинов*! $dolphin->accept($jump); // Появился над водой и исчез
```

Примеры на Java и Python.

## Стратегия (Strategy)

Википедия гласит:

> **Стратегия** — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

**Пример из жизни:** Возьмём пример с пузырьковой сортировкой. Мы её реализовали, но с ростом объёмов данных сортировка работа стала выполняться очень медленно. Тогда мы сделали быструю сортировку. 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 объёмов — быстрая.

**Простыми словами:** Шаблон стратегия позволяет переключаться между алгоритмами или стратегиями в зависимости от ситуации.

Перейдем к коду. Возьмем наш пример. Изначально у нас есть наша `SortStrategy` и разные её реализации:

```
interface SortStrategy { public function sort(array $dataset): array; } class BubbleSortStrategy implements SortStrategy { public function sort(array $dataset): array { echo "Сортировка пузырьком"; // Сортировка return $dataset; } } class QuickSortStrategy implements SortStrategy { public function sort(array $dataset): array { echo "Быстрая сортировка"; // Сортировка return $dataset; } }
```

И у нас есть `Sorter`, который собирается использовать какую-то стратегию:

```
class Sorter { protected $sorter; public function __construct(SortStrategy $sorter) { $this->sorter = $sorter; } public function sort(array $dataset): array { return $this->sorter->sort($dataset); } }
```

Пример использования:

```
$dataset = [1, 5, 4, 3, 2, 8]; $sorter = new Sorter(new BubbleSortStrategy()); $sorter->sort($dataset); // Вывод : Сортировка пузырьком $sorter = new Sorter(new QuickSortStrategy()); $sorter->sort($dataset); // Вывод : Быстрая сортировка
```

Примеры на Java и Python.

## Состояние (State)

Википедия гласит:

> **Состояние** — поведенческий шаблон проектирования. Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

**Пример из жизни:** Допустим, в графическом редакторе вы выбрали кисть. Она меняет своё поведение в зависимости от настройки цвета, т. е. рисует линию выбранного цвета.

**Простыми словами:** Шаблон позволяет менять поведение класса при изменении состояния.

Перейдем к примерам в коде. Возьмем пример текстового редактора, он позволяет вам менять состояние напечатанного текста. Например, если у вас выбран курсив, то он будет писать курсивом и так далее.

Изначально у нас есть интерфейс `WritingState` и несколько его реализаций:

```
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; } }
```

Затем `TextEditor`:

```
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); } }
```

Пример использования:

```
$editor = new TextEditor(new Default()); $editor->type('Первая строка'); $editor->setState(new UpperCase()); $editor->type('Вторая строка'); $editor->type('Третья строка'); $editor->setState(new LowerCase()); $editor->type('Четвертая строка'); $editor->type('Пятая строка'); // Output: // Первая строка // ВТОРАЯ СТРОКА // ТРЕТЬЯ СТРОКА // четвертая строка // пятая строка
```

Примеры на Java и Python.

## Шаблонный метод (Template Method)

Википедия гласит:

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

**Пример из жизни:** Допустим, вы собрались строить дома. Этапы будут такими:

1. Подготовка фундамента.
2. Возведение стен.
3. Настил крыши.
4. 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.

**Простыми словами:** Шаблонный метод определяет каркас выполнения определённого алгоритма, но реализацию самих этапов делегирует дочерним классам.

Обратимся к коду. Допустим, у нас есть программный инструмент, позволяющий тестировать, проводить контроль качества кода, выполнять сборку, генерировать отчёты сборки (отчёты о покрытии кода, о качестве кода и т. д.), а также развёртывать приложение на тестовом сервере.

Изначально у нас есть наш `Builder`, который описывает скелет для построения алгоритма:

```
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(); }
```

Затем у нас есть его реализации:

```
class AndroidBuilder extends Builder { public function test() { echo 'Запуск Android тестов'; } public function lint() { echo 'Копирование Android кода'; } public function assemble() { echo 'Android сборка'; } public function deploy() { echo 'Развертывание сборки на сервере'; } } class IosBuilder extends Builder { public function test() { echo 'Запуск iOS тестов'; } public function lint() { echo 'Копирование iOS кода'; } public function assemble() { echo 'iOS сборка'; } public function deploy() { echo 'Развертывание сборки на сервере'; } }
```

Пример использования:

```
$androidBuilder = new AndroidBuilder(); $androidBuilder->build(); // Вывод: // Запуск Android тестов // Копирование Android кода // Android сборка // Развертывание сборки на сервере $iosBuilder = new IosBuilder(); $iosBuilder->build(); // Вывод: // Запуск iOS тестов // Копирование iOS кода // iOS сборка // Развертывание сборки на сервере
```

Примеры на Java и Python.

Перевод статьи «Design Patterns for Humans»

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


**Также рекомендуем:**

Патриотичный цикл for(int c=0; c<10; c++)

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

Пользовательское соглашение
Политика конфиденциальности

«Аргументы и функции» — рассылка новостей
Включить уведомления