

6. Паттерны проектирования



Christofer Alexander. «A Pattern Language. Towns. Buildings. Constructions» 1977

- Идея **паттерна** как повторяемого решения конкретной проблемы в строительстве и городском планировании.
- Структурированный формат описания паттерна.
- "Язык паттернов", позволяющий создавать сложные системы из простых элементов.

Архитектурные паттерны

1. Сеть живых улиц (Network of Paths and Nodes)

- **Проблема** : Города часто страдают от монотонности и отсутствия социальной активности.
- **Решение** : Создание сети пешеходных улиц, площадей и общественных пространств, которые стимулируют случайные встречи и общение.
- **Пример** : Узкие улочки с кафе, лавками и скамейками, где люди могут взаимодействовать.

Архитектурные паттерны

2. Двор-улица (Street Café)

- Проблема : Отсутствие мест для неформального общения в городах.
- Решение : Интеграция открытых кафе и зон отдыха вдоль улиц, чтобы оживить общественное пространство.
- Пример : Парижские уличные кафе с столиками на тротуаре.

3. Краеугольное здание (Corner Building)

- Проблема : Пустующие углы зданий создают ощущение незавершенности.
- Решение : Акцентирование углов зданий с помощью декора, окон или функциональных элементов (например, входов).
- Пример : Здания с эркерами или арочными проемами на углах.



Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides
«Design Patterns. Elements of
Reusable Object-Oriented
Software» 1995

Концепция паттернов в ООП для превращения интуитивных решений в строгую методологию, задающую подходы к проектированию ПО

Зачем нужны паттерны в ООП?

Шаблоны упрощают проектирование и поддержку программ.

- **Проверенные решения.**

Ваш код более предсказуем когда вы используете готовые решения, вместо повторного изобретения велосипеда.

- **Стандартизация кода.**

Использование типовых унифицированных решений — должно быть меньше ошибок.

- **Общий язык.**

По названию шаблона понятно, какой подход вы выбрали и какие классы для этого нужны.

Типы шаблонов

- **Порождающие шаблоны** — управляют созданием объектов без внесения в программу лишних зависимостей.
- **Структурные шаблоны** — показывают различные способы построения связей между объектами.
- **Поведенческие шаблоны** — для эффективной коммуникации между объектами.

Порождающие шаблоны

- Factory method
- Abstract factory
- Singleton
- ...

Структурные шаблоны

- Adapter
- Decorator
- ...

Поведенческие шаблоны

- Observer
- Strategy



На третьем ходу выяснилось, что гроссмейстер играет восемнадцать **испанских партий**. В остальных двенадцати черные применили хотя и устаревшую, но довольно верную **защиту Филидора**. Если б Остап узнал, что он играет такие мудреные партии и сталкивается с такой испытанной защитой, он крайне бы удивился. Дело в том, что великий комбинатор играл в шахматы второй раз в жизни.

И. Ильф, Е. Петров «Двенадцать стульев»

Simple factory

Simple factory

Паттерн Simple Factory («Простая фабрика») решает проблему сложности управления созданием объектов в коде, особенно когда:

1. Много условий для выбора типа создаваемого объекта.
2. Дублирование кода в разных частях программы.
3. Жесткая зависимость клиентского кода от конкретных классов.

php

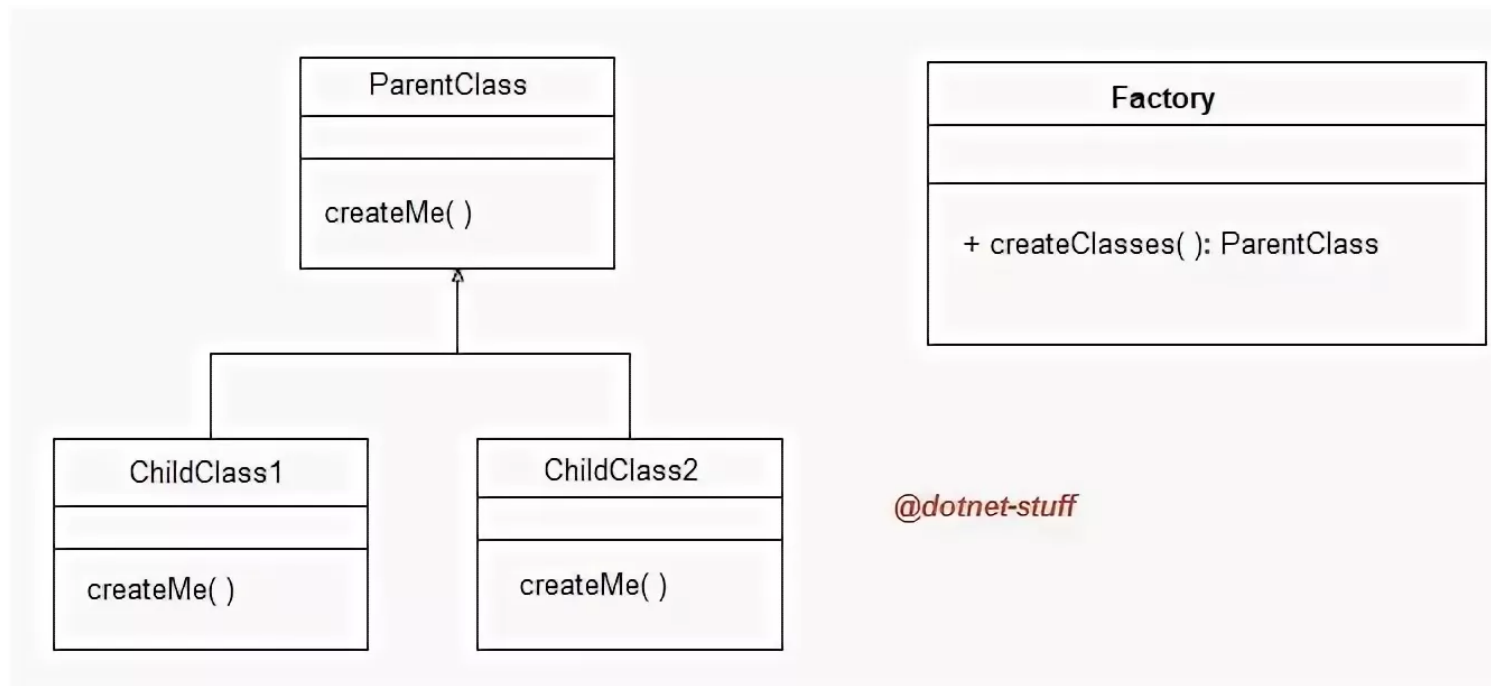
```
1 if ($type == 'dog') {  
2     $animal = new Dog();  
3 } elseif ($type == 'cat') {  
4     $animal = new Cat();  
5 } elseif ($type == 'bird') {  
6     $animal = new Bird();  
7 }
```

Simple factory

Фабрика — объект или класс, создающий другие объекты.
Простая фабрика генерирует экземпляр класса для клиента.

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.



Simple factory

```
interface User {
    public function getRole(): string;
}

class Admin implements User {
    public function getRole(): string {
        return "Администратор";
    }
}

class Guest implements User {
    public function getRole(): string {
        return "Гость";
    }
}

class UserFactory {
    public static function createUser(string $role): User {
        switch ($role) {
            case 'admin':
                return new Admin();
            case 'guest':
                return new Guest();
            default:
                throw new Exception("Неизвестная роль");
        }
    }
}

// Использование
$user = UserFactory::createUser('admin');
echo $user->getRole(); // Вывод: "Администратор"
```

Simple factory

Плюсы :

- Убирает дублирование кода создания объектов.
- Клиентский код не зависит от конкретных классов.

Минусы :

- Нарушает принцип открытости/закрытости (ОСР): при добавлении нового продукта нужно менять код фабрики.
- Может стать "божественным объектом" при чрезмерном усложнении.

Factory method

The "Factory" Model

- A design pattern used to implement the services.



Raw
Materials

Input data

Car
Manufacturing
Service



Suzuki Car Factory



Honda Car Factory



BMW Car Factory



Определяет общий интерфейс для создания объектов в суперклассе, но позволяет подклассам изменять тип создаваемых объектов.

Когда использовать?

- Заранее неизвестно, объекты каких типов необходимо создавать
- Система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать
- Создание новых объектов необходимо делегировать из базового класса классам наследникам

Структура Factory method

1. **Продукт** — общий интерфейс для всех создаваемых объектов.
 2. **Конкретные продукты** — классы, реализующие интерфейс продукта.
 3. **Создатель (Creator)** — абстрактный класс, который объявляет фабричный метод (`factoryMethod()`), возвращающий объект продукта.
 4. **Конкретные создатели** — наследники Creator, реализующие фабричный метод для создания конкретных продуктов.
-

1. Продукт (интерфейс)

php

```
1 interface Document {  
2     public function generate(): string;  
3 }
```

2. Конкретные продукты

php

```
1 class PDFDocument implements Document {  
2     public function generate(): string {  
3         return "Генерация PDF документа";  
4     }  
5 }  
6  
7 class WordDocument implements Document {  
8     public function generate(): string {  
9         return "Генерация Word документа";  
10    }  
11 }
```

3. Создатель (Creator)

```
php
1 abstract class DocumentCreator {
2     // Фабричный метод (абстрактный)
3     abstract public function factoryMethod(): Document;
4
5     // Бизнес-логика, которая использует продукт
6     public function renderDocument(): string {
7         $document = $this->factoryMethod();
8         return $document->generate();
9     }
10 }
```

4. Конкретные создатели

```
php
1 class PDFDocumentCreator extends DocumentCreator {
2     public function factoryMethod(): Document {
3         return new PDFDocument();
4     }
5 }
6
7 class WordDocumentCreator extends DocumentCreator {
8     public function factoryMethod(): Document {
9         return new WordDocument();
10    }
11 }
```

5. Использование

```
php
1 function clientCode(DocumentCreator $creator) {
2     echo $creator->renderDocument();
3 }
4
5 // Создаем PDF
6 $pdfCreator = new PDFDocumentCreator();
7 clientCode($pdfCreator); // Вывод: "Генерация PDF документа"
8
9 // Создаем Word
10 $wordCreator = new WordDocumentCreator();
11 clientCode($wordCreator); // Вывод: "Генерация Word документа"
```

Factory method

Плюсы :

- Соответствует принципу OCP: добавление новых продуктов не требует изменения существующего кода.
- Убирает прямую зависимость между клиентским кодом и конкретными классами продуктов.

Минусы :

- Может привести к созданию большого числа подклассов.
- Не всегда удобен, если требуется создавать объекты с разными параметрами.

Отличие от Simple Factory

- **Simple Factory** использует статический метод для создания объектов, нарушая OCP (при добавлении нового продукта нужно менять код фабрики).
- **Factory Method** использует наследование: каждый новый продукт требует создания нового подкласса Creator.

Builder

Builder

- Позволяет создавать сложные объекты пошагово.
- Отделяет процесс конструирования объекта от его представления, позволяя использовать один и тот же процесс для создания различных представлений объекта.

Применимость Builder

- Упрощает создание объектов с большим количеством параметров , особенно если многие из них опциональны.
- Избегает "телескопического конструктора" (множества перегруженных конструкторов).
- Позволяет создавать объекты с валидацией и настройкой на каждом этапе.

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

Структура Builder

Структура Builder

1. **Продукт** — сложный объект, который создается.
2. **Builder** — абстрактный класс или интерфейс, объявляющий шаги построения продукта.
3. **Concrete Builder** — реализует шаги построения конкретного продукта.
4. **Director** (опционально) — управляет процессом построения, используя Builder.

Пример 1: Создание объекта User

Допустим, у нас есть класс `User` с множеством полей, многие из которых опциональны.

1. Продукт

```
class User {
    private string $name;
    private string $email;
    private ?string $address = null;
    private ?string $phone = null;

    // Приватный конструктор, чтобы запретить прямое создание
    private function __construct() {}

    public static function builder(): UserBuilder {
        return new UserBuilder();
    }

    // Геттеры...
}
```

2. Builder

```
class UserBuilder {
    private User $user;

    public function __construct() {
        $this->user = new User();
    }

    public function setName(string $name): self {
        $this->user->name = $name;
        return $this;
    }

    public function setEmail(string $email): self {
        $this->user->email = $email;
        return $this;
    }

    public function setAddress(?string $address): self {
        $this->user->address = $address;
        return $this;
    }

    public function setPhone(?string $phone): self {
        $this->user->phone = $phone;
        return $this;
    }
}
```

2. Builder

```
public function setPhone(?string $phone): self {  
    $this->user->phone = $phone;  
    return $this;  
}
```

```
public function build(): User {  
    // Можно добавить валидацию  
    if (empty($this->user->name)) {  
        throw new Exception("Имя обязательно");  
    }  
    return $this->user;  
}
```

3. Использование

php

```
1 $user = User::builder()  
2     ->setName("Иван")  
3     ->setEmail("ivan@example.com")  
4     ->setPhone("+71234567890")  
5     ->build();
```

Builder

Плюсы :

- Устраняет необходимость в множестве конструкторов.
- Позволяет создавать объекты с опциональными параметрами.
- Упрощает код инициализации сложных объектов.

Минусы :

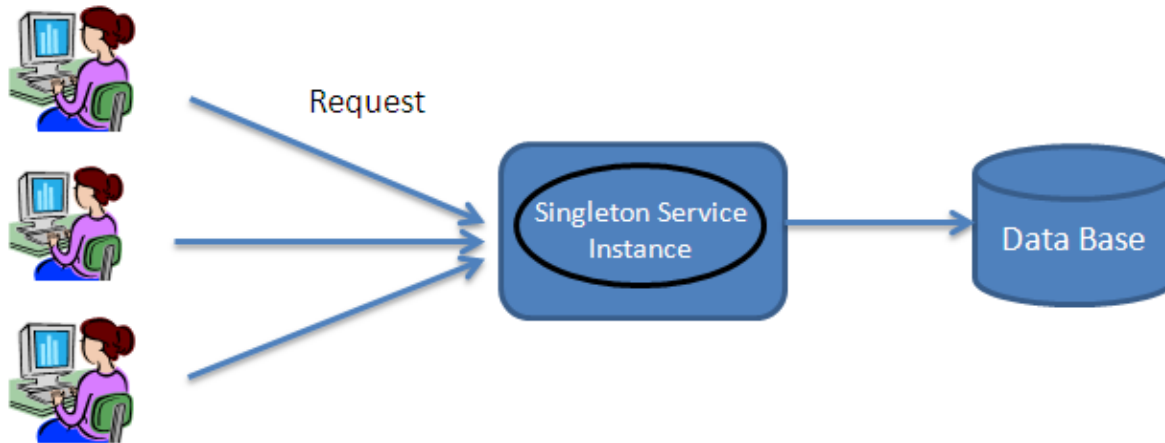
- Усложняет код для простых объектов.
- Требуется создание дополнительных классов (Builder, Director).

В PHP часто используется **Fluent Interface** (цепочка методов) для реализации Builder.

Singleton

Singleton

Гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к нему.



Singleton
-instance : Singleton
-Singleton() +getInstance() : Singleton

Структура Singleton

1. Приватный конструктор — запрещает создание объектов через `new`.
2. Статическое свойство — хранит единственный экземпляр класса.
3. Статический метод (например, `getInstance()`) — возвращает экземпляр.

Singleton

```
class Singleton {
    // Статическое свойство для хранения экземпляра
    private static ?Singleton $instance = null;

    // Приватный конструктор
    private function __construct() {}

    // Метод для получения экземпляра
    public static function getInstance(): Singleton {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    // Запрет клонирования
    private function __clone() {}
    private function __wakeup() {}
}

// Использование
$instance1 = Singleton::getInstance();
$instance2 = Singleton::getInstance();

var_dump($instance1 === $instance2); // true (один и т
```

Singleton

```
class Logger {
    private static ?Logger $instance = null;
    private array $logs = [];

    private function __construct() {}

    public static function getInstance(): Logger {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    public function log(string $message): void {
        $this->logs[] = $message;
    }

    public function getLogs(): array {
        return $this->logs;
    }

    private function __clone() {}
    private function __wakeup() {}
}
```

```
// Использование
$logger = Logger::getInstance();
$logger->log("Ошибка: что-то пошло не так");
```

Singleton

Плюсы :

- Гарантирует единственный экземпляр.
- Удобен для управления общими ресурсами (базы данных, логгеры).
- Ленивая инициализация (экземпляр создается только при первом запросе).

Минусы :

- Нарушает **принцип единственной ответственности (SRP)** — управляет жизненным циклом и решает свою задачу.
- Сложно тестировать (глобальное состояние может вызвать побочные эффекты).
- Может создать проблемы в многопоточной среде (в PHP это менее актуально).

Singleton

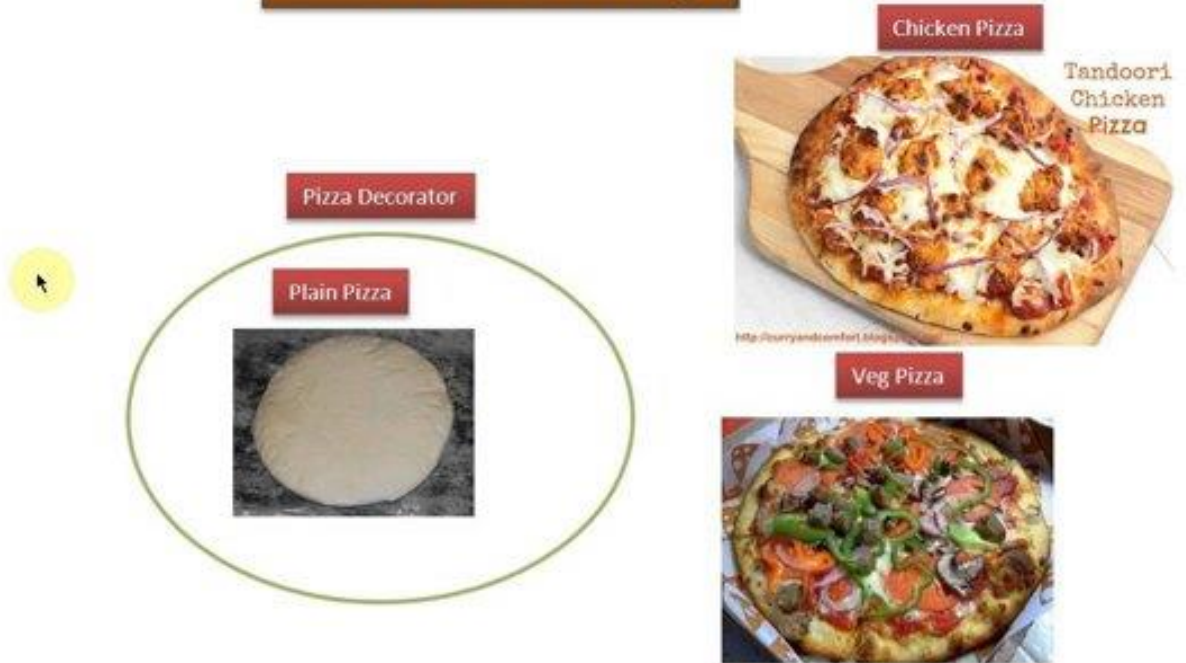
Singleton полезен в случаях, где:

- Требуется строго один экземпляр класса.
- Нужна глобальная точка доступа к объекту.
- Ресурсы должны быть инициализированы единожды.

Однако его следует использовать с осторожностью, так как глобальное состояние усложняет тестирование и поддержку кода. В современной разработке часто предпочитают **Dependency Injection** вместо Singleton.

Decorator

Decorator Pattern – Real Time Example



Decorator

Позволяет подключать к объекту дополнительное поведение (статически или динамически), не влияя на поведение других объектов того же класса.

- **Альтернатива наследованию** : позволяет добавлять функциональность без создания множества подклассов.
- **Гибкость** : декораторы можно комбинировать, оборачивая объекты в несколько слоёв.
- **Соблюдение SRP** : каждый декоратор отвечает за свою часть функциональности.

Структура Decorator

1. **Компонент (Component)** — общий интерфейс для объектов и декораторов.
2. **Конкретный компонент (Concrete Component)** — базовый объект, который будет декорироваться.
3. **Декоратор (Decorator)** — абстрактный класс, реализующий интерфейс Component и хранящий ссылку на него.
4. **Конкретные декораторы (Concrete Decorators)** — добавляют новую функциональность.

Пример 1: Форматирование текста

Допустим, мы хотим динамически добавлять стили к тексту (жирный, курсив, подчеркивание).

1. Компонент (интерфейс)

php

Копировать

```
1 interface TextComponent {
2     public function render(): string;
3 }
```

2. Конкретный компонент

php

Копировать

```
1 class PlainText implements TextComponent {
2     private string $text;
3
4     public function __construct(string $text) {
5         $this->text = $text;
6     }
7
8     public function render(): string {
9         return $this->text;
10    }
11 }
```

3. Базовый декоратор

php

Копировать

```
1 abstract class TextDecorator implements TextComponent {
2     protected TextComponent $component;
3
4     public function __construct(TextComponent $component)
5         $this->component = $component;
6     }
7 }
```

4. Конкретные декораторы

php

Копировать

```
1 class BoldDecorator extends TextDecorator {
2     public function render(): string {
3         return "<b>" . $this->component->render() . "</b>";
4     }
5 }
6
7 class ItalicDecorator extends TextDecorator {
8     public function render(): string {
9         return "<i>" . $this->component->render() . "</i>";
10    }
11 }
12
13 class UnderlineDecorator extends TextDecorator {
14     public function render(): string {
15         return "<u>" . $this->component->render() . "</u>";
16    }
17 }
```

5. Использование

```
php
1 $text = new PlainText("Hello, World!");
2
3 // Оборачиваем в декораторы
4 $decoratedText = new BoldDecorator(
5     new ItalicDecorator(
6         new UnderlineDecorator($text)
7     )
8 );
9
10 echo $decoratedText->render();
11 // Вывод: <b><i><u>Hello, World!</u></i></b>
```

Decorator

Плюсы :

- Гибкость: можно комбинировать декораторы в любом порядке.
- Соответствует принципу открытости/закрытости (ОСР).
- Избегает взрывного роста подклассов.

Минусы :

- Может привести к созданию множества мелких классов.
- Сложность отладки из-за вложенных декораторов.

Decorator

Decorator полезен, когда:

- Нужно динамически добавлять обязанности объектам.
- Хочется избежать наследования для расширения функциональности.
- Требуется гибкость в комбинации функций.

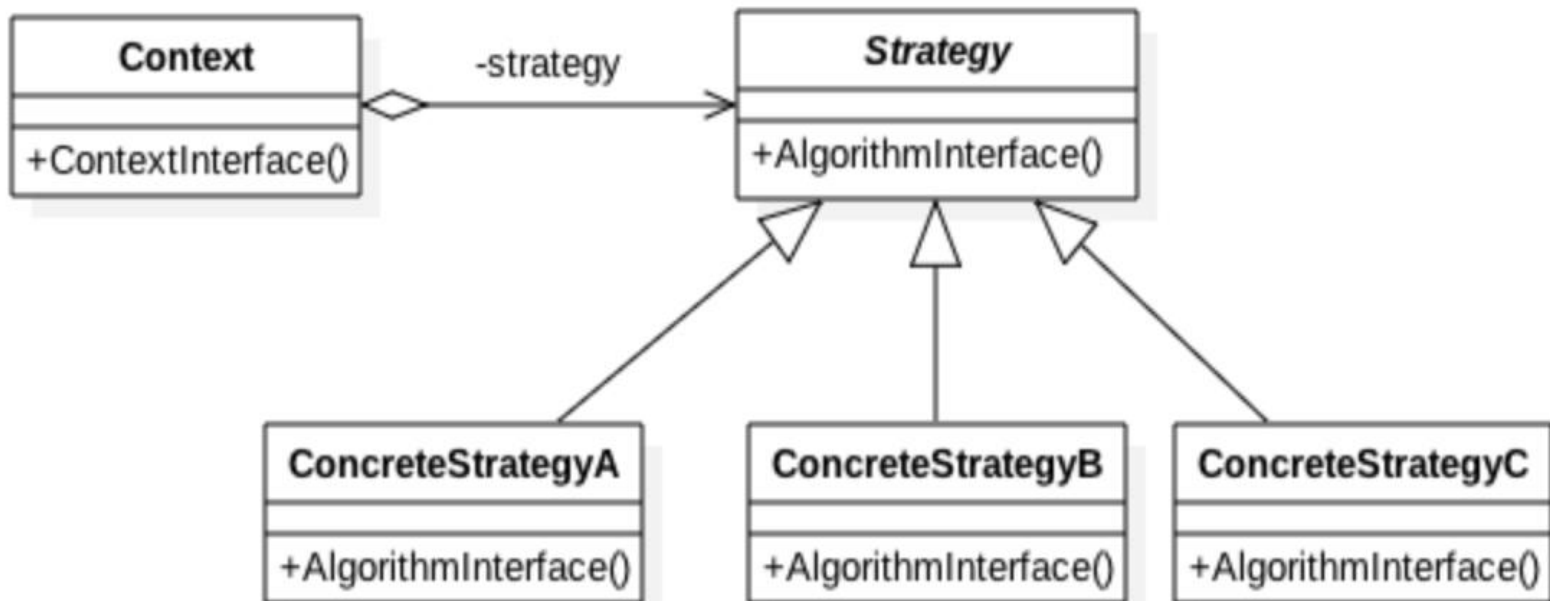
В PHP этот паттерн часто используется в фреймворках (например, для `middleware` или обработки HTTP-запросов).

Strategy

Strategy

Позволяет переключаться между алгоритмами или стратегиями в зависимости от ситуации (независимо от клиентов, их использующих).

Альтернатива наследованию (вместо расширения абстрактного класса).



Зачем нужен Strategy

- Избегает множества условных операторов (`if-else` или `switch`), заменяя их полиморфизмом.
- Инкапсулирует алгоритмы , делая их независимыми от клиентского кода.
- Позволяет динамически менять поведение объекта.

Структура Strategy

1. Интерфейс стратегии — общий интерфейс для всех конкретных стратегий.
2. Конкретные стратегии — реализуют интерфейс стратегии.
3. Контекст — использует стратегию, делегируя ей выполнение задачи.

Strategy

Плюсы :

- Упрощает добавление новых алгоритмов.
- Избавляет от дублирования кода.
- Позволяет динамически менять поведение объекта.

Минусы :

- Увеличивает количество классов.
- Клиентский код должен знать о разных стратегиях.

В PHP этот паттерн часто используется в фреймворках (например, для валидации, логирования или обработки платежей).

Strategy и Factory method

Оба паттерна **Strategy** и **Factory Method** относятся к поведенческим и порождающим паттернам проектирования соответственно, но у них есть общие черты:

1. Инкапсуляция логики :

- Оба паттерна инкапсулируют логику в отдельные классы, что делает код более модульным и поддерживаемым.
- В случае **Strategy** , инкапсулируется алгоритм (поведение).
- В случае **Factory Method** , инкапсулируется процесс создания объектов.

Strategy и Factory method

2. Полиморфизм :

- Оба паттерна активно используют полиморфизм. Они работают с интерфейсами или абстрактными классами, что позволяет легко добавлять новые реализации без изменения существующего кода.

3. Расширяемость :

- Благодаря использованию интерфейсов или абстракций, оба паттерна позволяют легко расширять функциональность системы, добавляя новые стратегии или фабрики.

Strategy и Factory method

4. Устранение жесткой зависимости :

- Оба паттерна помогают избежать жесткой привязки к конкретным реализациям. Вместо этого они предоставляют механизм для динамического выбора нужной реализации.

Ответ : Общие черты **Strategy** и **Factory Method** заключаются в инкапсуляции логики, использовании полиморфизма, расширяемости и устранении жесткой зависимости.

Facade

Facade

- Предоставляет упрощенный интерфейс к сложной системе классов, библиотеке или подсистеме.
- Скрывает сложность системы за простым API, делая её использование более удобным для клиентского кода.

Зачем нужен Facade?

- Упрощает взаимодействие с сложной подсистемой.
- Снижает зависимость клиента от конкретных классов системы.
- Разделяет ответственность: клиенту не нужно знать о внутреннем устройстве подсистемы.

Структура Facade

1. **Подсистемные классы** — набор классов, реализующих функциональность системы.
2. **Facade** — класс, который предоставляет простой интерфейс для работы с подсистемными классами.
3. **Клиент** — использует Facade для взаимодействия с подсистемой.

Система отправки уведомлений через email, SMS и Push-уведомления.

1. Подсистемные классы

php

Копировать

```
1 class EmailSender {
2     public function sendEmail(string $email, string $message): void {
3         echo "Отправка email на $email: $message\n";
4     }
5 }
6
7 class SMSSender {
8     public function sendSMS(string $phone, string $message): void {
9         echo "Отправка SMS на $phone: $message\n";
10    }
11 }
12
13 class PushSender {
14     public function sendPush(string $deviceId, string $message): void {
15         echo "Отправка Push на устройство $deviceId: $message\n";
16     }
17 }
```

2. Facade

```
php
1 class NotificationFacade {
2     private EmailSender $emailSender;
3     private SMSSender $smsSender;
4     private PushSender $pushSender;
5
6     public function __construct() {
7         $this->emailSender = new EmailSender();
8         $this->smsSender = new SMSSender();
9         $this->pushSender = new PushSender();
10    }
11
12    // Упрощенный метод для массовой отправки
13    public function sendAllNotifications(
14        string $email,
15        string $phone,
16        string $deviceId,
17        string $message
18    ): void {
19        $this->emailSender->sendEmail($email, $message);
20        $this->smsSender->sendSMS($phone, $message);
21        $this->pushSender->sendPush($deviceId, $message);
22    }
23 }
```

3. Использование

```
php
1 $facade = new NotificationFacade();
2 $facade->sendAllNotifications(
3     "user@example.com",
4     "+71234567890",
5     "device_123",
6     "Ваш заказ готов!"
7 );
8 // Вывод:
9 // Отправка email на user@example.com: Ваш заказ готов!
10 // Отправка SMS на +71234567890: Ваш заказ готов!
11 // Отправка Push на устройство device_123: Ваш заказ готов!
```

Facade

Плюсы :

- Упрощает использование сложных систем.
- Снижает зависимость клиента от подсистемы.
- Улучшает читаемость и поддерживаемость кода.

Минусы :

- Может стать "божественным объектом" (God Object), если охватывает слишком много функций.
- Может нарушить принцип единой ответственности (Single Responsibility Principle).

В PHP Facade часто используется в фреймворках (например, Laravel) для упрощения доступа к сложным функциям (базы данных, очереди, кэш).

Adapter

Adapter

- Позволяет объектам с несовместимыми интерфейсами работать вместе.
- Выступает «переводчиком» между двумя интерфейсами, преобразуя запросы одного класса в формат, понятный другому.

Зачем нужен Adapter?

- Интеграция старого и нового кода : когда нужно использовать legacy-классы с новым API.
- Совместимость библиотек : когда сторонние библиотеки имеют разные интерфейсы.
- Расширение функционала : добавление нового функционала без изменения существующего кода.

Структура Adapter

1. **Целевой интерфейс (Target)** — интерфейс, который ожидает клиентский код.
2. **Адаптируемый класс (Adaptee)** — класс с несовместимым интерфейсом.
3. **Адаптер (Adapter)** — класс, который реализует целевой интерфейс и содержит ссылку на Adaptee.

Пример 1: Работа с разными форматами данных

Допустим, у нас есть класс для работы с XML, но клиентский код ожидает данные в JSON.

1. Целевой интерфейс (JSON)

php

Копиров


```
1 interface JsonData {
2     public function getJson(): string;
3 }
```

2. Адаптируемый класс (XML)

php

Копиров

```
1 class XmlData {
2     private $data;
3
4     public function __construct($data) {
5         $this->data = $data;
6     }
7
8     public function getXml(): string {
9         return "<data>{$this->data}</data>";
10    }
11 }
```



3. Адаптер

```
php Kon  
1 class XmlToJsonAdapter implements JsonData {  
2     private $xmlData;  
3  
4     public function __construct(XmlData $xmlData) {  
5         $this->xmlData = $xmlData;  
6     }  
7  
8     public function getJson(): string {  
9         // Преобразуем XML в JSON  
10        $xml = simplexml_load_string($this->xmlData->getXml());  
11        return json_encode($xml);  
12    }  
13 }
```

4. Использование

```
php Kon  
1 $xmlData = new XmlData("Hello, Adapter!");  
2 $adapter = new XmlToJsonAdapter($xmlData);  
3  
4 echo $adapter->getJson(); // {"data":"Hello, Adapter!"}
```

Плюсы :

- Позволяет использовать несовместимые классы вместе.
- Сохраняет принцип открытости/закрытости (ОСР).
- Упрощает интеграцию сторонних библиотек.

Минусы :

- Усложняет код из-за дополнительных классов.
 - Может привести к избыточным адаптерам при плохом проектировании.
-

Отличие от других паттернов

- **Decorator** : Добавляет функционал, не меняя интерфейс. Adapter изменяет интерфейс.
- **Facade** : Упрощает сложную систему, а Adapter адаптирует интерфейс.

В PHP Adapter часто используется для работы с разными API платежных систем, баз данных или внешних сервисов.

Не усложняйте!

Не усложняйте

- **KISS (Keep It Simple, Stupid)**. Пишите проще.
- **YAGNI (You Aren't Gonna Need It)**. Вам это не понадобится. Не добавляйте функциональность, пока она не понадобится прямо сейчас.

KISS в ООП

KISS в ООП — это:

- Отказ от избыточных абстракций.
- Предпочтение композиции перед наследованием.
- Чёткая ответственность классов и методов.
- Ясность вместо «умных» решений.

Простой код легче тестировать, поддерживать и расширять, даже если это требует небольших компромиссов с «идеальной» архитектурой.

Не усложняйте

- **YAGNI (You Aren't Gonna Need It)**. Вам это не понадобится. Не добавляйте функциональность, пока она не понадобится прямо сейчас.

Зачем нужен YAGNI?

- Упрощает код : меньше кода → меньше ошибок.
- Экономит время : не тратите ресурсы на ненужные фичи.
- Снижает технический долг : избегает сложных абстракций «на потом».

Нарушение YAGNI

Предположим, вы разрабатываете систему логирования и сразу создаете абстракцию для «будущих» логгеров:

```
php Копировать
1 // Лишняя абстракция "на будущее"
2 abstract class Logger {
3     abstract public function log(string $message);
4 }
5
6 class FileLogger extends Logger {
7     public function log(string $message) {
8         // Логирование в файл
9     }
10 }
11
12 // "А вдруг нам понадобится логировать в базу?"
13 class DatabaseLogger extends Logger {
14     public function log(string $message) {
15         // Пока не реализовано
16     }
17 }
```

Проблема : Код для `DatabaseLogger` написан заранее, но его никто не использует. Это усложняет поддержку.