Разработка многостраничного сайта на PHP

ТЕМА 4.1 АРХИТЕКТУРА ПРИЛОЖЕНИЙ И ШАБЛОНЫ ПРОЕКТИРОВАНИЯ ЗАНЯТИЯ № 1 - ЛЕКЦИЯ

Тема занятия – Архитектура приложений и шаблоны проектирования

Цель занятия –

Изучить существующие шаблоны и виды архитектуры веб-приложений

Актуализация

В прошлом модули мы изучили HTTP – протоколы, поработали с формами, выяснили, что такое объектно-ориентированное программирование и, конечно, много практиковались.

В заключительном модуле нам предстоит изучить :

- Архитектуру приложений, а именно, какие есть виды архитектуры, какие есть преимущества и недостатки их использования,
- Шаблоны проектирования, а именно какие существуют методы, чем они различаются, какие есть плюсы и минусы их использования, и, конечно же, рассмотрим их на практических примерах.

Содержание занятия

1) Введение

2) Архитектура приложений на PHP (Монолитная архитектура, Микросервисная архитектура, MVC-архитектура)

3) Шаблоны проектирования на PHP (Фабричный метод , Одиночка, Стратегия, Адаптер, Декоратор, Шаблонный метод)

4) Заключение

5) Список используемой литературы

Введение

Архитектура приложений и шаблоны проектирования - это базовые концепции, которые используются при разработке сложных программных решений на PHP.

Хорошо известное приложение позволяет легко добавлять новые функции, повышать производительность и сокращать время разработки.

Шаблоны проектирования, в свою очередь, создают ресурсы и гибкие архитектуры, которые могут быть переиспользованы для различных приложений.

Архитектура приложения

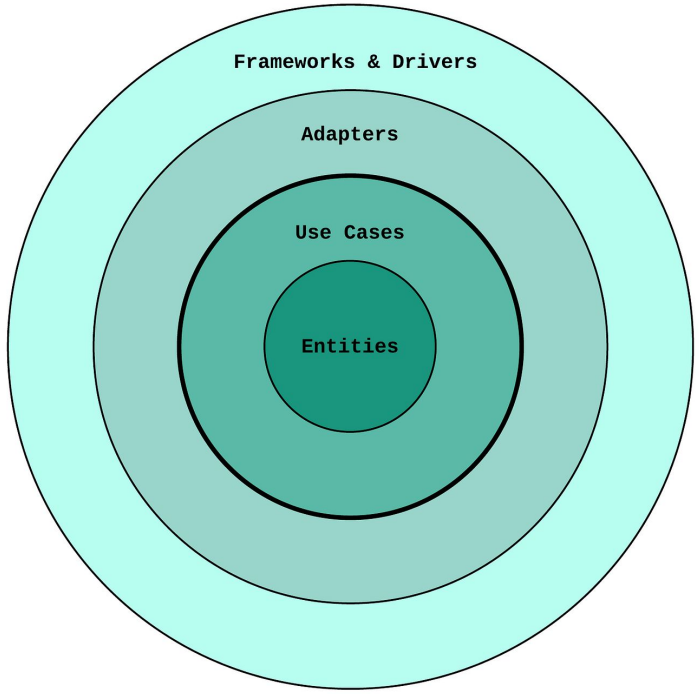
Что же такое эта архитектура?

Архитектура программного обеспечения (англ. software architecture)—

совокупность важнейших решений

об организации программной

системы.



Архитектура приложения

Архитектура включает:

- выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;
- соединение выбранных элементов структуры;
- архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

Архитектура приложения

Если говорить проще, то архитектура - это про то, как:

- разделить приложение на какие-то блоки;
- разложить эти блоки по своим местам;
- связать эти блоки между собой.

Архитектура приложения

Документирование архитектуры программного обеспечения (ПО) упрощает процесс коммуникации между разработчиками, позволяет зафиксировать принятые проектные решения и предоставить информацию о них эксплуатационному персоналу системы, повторно использовать компоненты и шаблоны проекта в других проектах.

Архитектура приложения

Основополагающей идеей дисциплины программной архитектуры является идея снижения сложности системы с помощью абстракции и разграничения полномочий.

На сегодняшний день до сих пор нет единого определения термина «архитектура программного обеспечения».

Так, сайт Института Программной Инженерии приводит более 150 определений этого понятия.

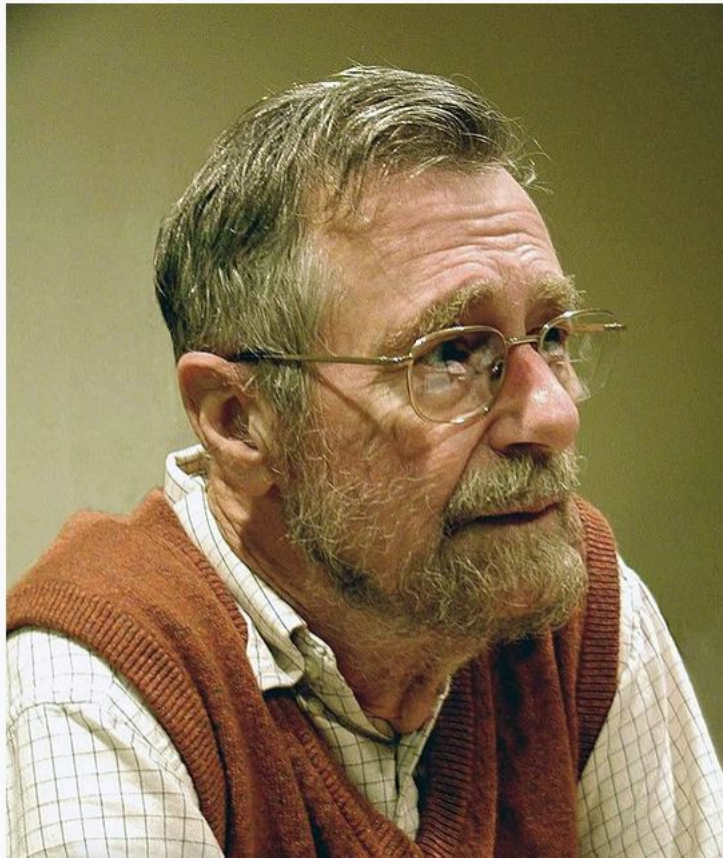
История

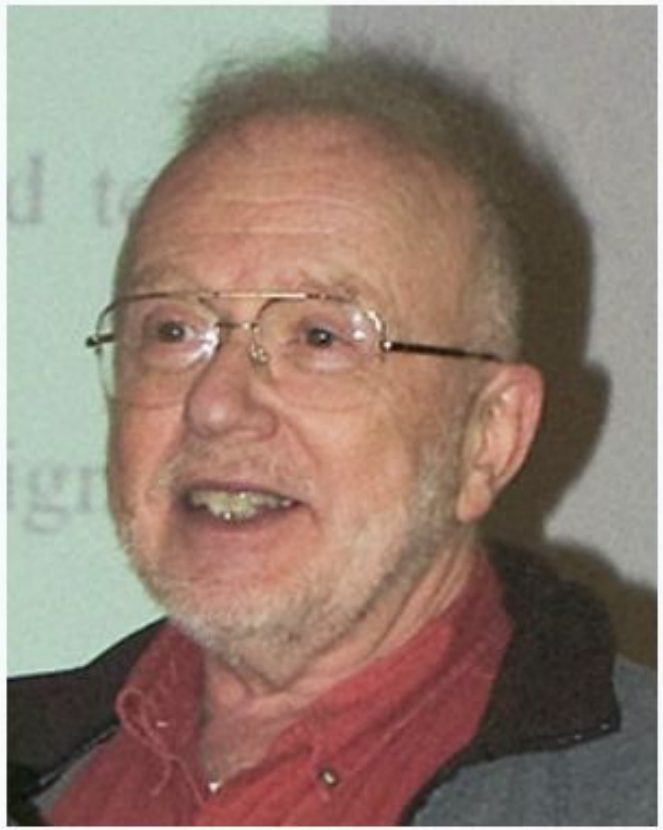
Поговорим немного об истории:

Начало архитектуры программного обеспечения как концепции было положено в научно-исследовательской работе Эдсгера Дейкстры в 1968 году и Дэвида Парнаса в начале 1970-х.

Эти ученые подчеркнули, что структура системы ПО имеет важное значение, и что построение правильной структуры — критически важно.

Эдсгер Дейкстр и Дэвид Парнас





История

Популярность изучения этой области возросла с начала 1990-х годов вместе с научно-исследовательской работой по исследованию архитектурных стилей (шаблонов), языков описания архитектуры, документирования архитектуры, и формальных методов.

В развитии архитектуры программного обеспечения как дисциплины играют важную роль научно-исследовательские учреждения.

История

- Мэри Шоу и Дэвид Гэрлан из университета Carnegie Mellon написали книгу под названием «Архитектура программного обеспечения: перспективы новой дисциплины» в 1996 году, в которой выдвинули концепции архитектуры программного обеспечения, такие как компоненты, соединители (connectors), стили и так далее.
- В калифорнийском университете Ирвайна по исследованию ПО в первую очередь исследует архитектурные стили, языки описания архитектуры и динамические архитектуры.

Языки описания архитектуры

Различными организациями было разработано несколько различных ADLS, в том числе:

- AADL (стандарт SAE),
- Wright (разработан в университете Carnegie Mellon),
- Acme (разработан в университете Carnegie Mellon),
- xADL (разработан в UCI),
- Darwin (разработан в Imperial College в Лондоне),
- DAOP-ADL (разработан в Университете Малаги),
- а также ByADL (Университет L'Aquila, Италия).

Общими элементами для всех этих языков являются понятия компонента, коннектора и конфигурации. Также, помимо специализированных языков, для описания архитектуры часто используется унифицированный язык моделирования UML.

Архитектура приложений

Являясь в настоящий момент своего развития дисциплиной без четких правил о «правильном» пути создания системы, проектирование архитектуры ПО все ещё является смесью науки и искусства. Аспект «искусства» заключается в том, что любая коммерческая система подразумевает наличие применения или миссии.

Архитектура приложений

С точки зрения пользователя программной архитектуры, программная архитектура дает направление для движения и решения задач, связанных со специальностью каждого такого пользователя. Например,

- заинтересованного лица,
- разработчика ПО,
- группы поддержки ПО,
- специалиста по сопровождению ПО,
- специалиста по развертыванию ПО,
- тестера,
- а также конечных пользователей.

Архитектура приложений

В этом смысле архитектура программного обеспечения на самом деле объединяет различные точки зрения.

Тот факт, что эти несколько различных точек зрения могут быть объединены в архитектуре программного обеспечения, является аргументом в защиту необходимости и целесообразности создания архитектуры ПО ещё до этапа разработки ПО.

Архитектура приложений

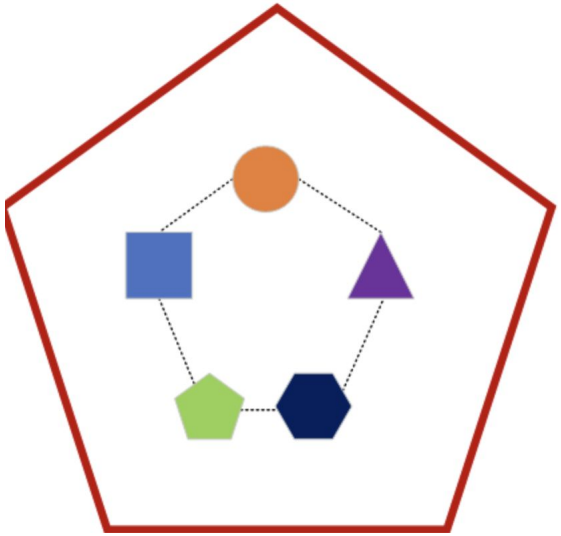
Существует 3 основных типа архитектуры приложений:

1. Монолитная архитектура – показывает систему как структуру из различных программных блоков.
2. Микросервисная архитектура – показывает размещение элементов системы во внешних средах.
3. MVC - архитектура – показывает систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов).

Модульная(монолитная) архитектура

Монолитная архитектура — это классический подход к разработке веб-приложений на PHP, где все компоненты приложений находятся в одной кодовой базе.

В монолитной архитектуре весь код приложения обычно разбит на модули или слои, такие как пользовательский интерфейс, бизнес-логика, доступ к данным и т.д.



Преимущества

- Простота разработки и развертывания: все компоненты находятся в одной кодовой базе, что облегчает процесс разработки и развертывания приложения.
- Простота масштабирования: монолитная архитектура позволяет просто добавлять новые серверы или ресурсы, чтобы обеспечить масштабируемость приложений.
- Простота тестирования: тестирование монолитного приложения проще, чем тестирование системы.

Недостатки

- Ограниченная масштабируемость: монолитное приложение имеет ограниченные возможности для масштабирования, так как все компоненты находятся в кодовой базе.
- Сложность поддержки: с ростом размера приложения могут возникнуть проблемы с поддержкой и развитием кода.
- Одиночная точка отказа: если монолитное приложение перерабатывается, то весь сайт останавливается.

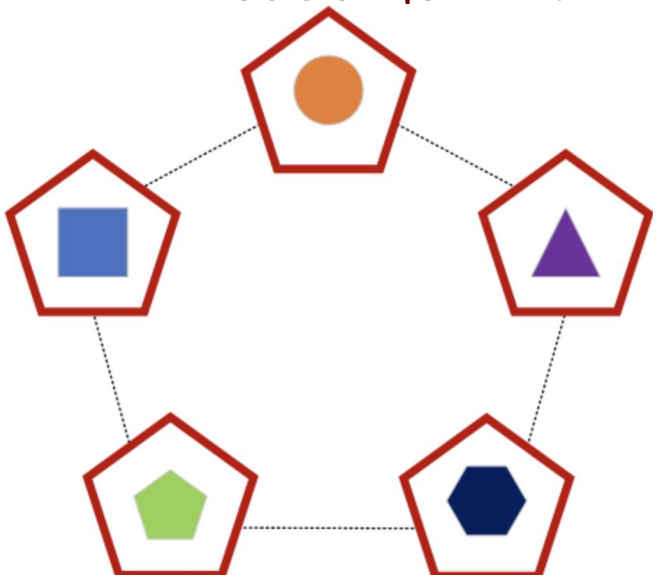
Вывод:

Монолитная архитектура на PHP является хорошим выбором для малых и средних проектов, где не требуется высокая масштабируемость и сложность разработки.

Она также может быть полезна для быстрого прототипирования и начала разработки.

Микросервисная архитектура

Микросервисная архитектура на PHP — это подход к разработке веб-приложений банка, который при приложении разбивается на отдельные сервисы, которые разработаны независимо друг от друга. Каждый сервис выполняет свою узкоспециализированную работу с другими сервисами через API или сообщения.



Преимущества

- Гибкость: сервисы могут быть внедрены и установлены независимо друг от друга, что позволяет быстро внедрять новые функции и изменения.
- Масштабируемость: каждый сервис может масштабироваться независимо друг от друга, что позволяет добиться высокой производительности и отказоустойчивости приложений.
- Удобство поддержки: каждый сервис легко поддерживается и изменяется, так как его код содержит только необходимый функционал.
- Легкость разработки: каждый сервис может быть разработан с использованием различных технологий и языков программирования, что позволяет использовать самые подходящие инструменты для каждой задачи.

Недостатки

- Сложность разработки и оценки: при сборе микроархитектуры требуется разработка и объединение большого количества ресурсов, что может требовать расхода значительных результатов.
- Нагрузка на сеть: микросервисная архитектура требует большую пропускную способность сети и может привести к снижению нагрузки на сеть.
- Сложность тестирования: тестирование микросервисной архитектуры сложнее, чем тестирование монолитной архитектуры, так как требуется тестирование каждого сервера отдельно.

Вывод:

Микросервисная архитектура на PHP является хорошим выбором для крупных проектов, где требуется высокая масштабируемость и развитие.

Она также может быть полезна для проектов, где требуется использование различных технологий и языков программирования.

MVC - архитектура

MVC (Model-View-Controller) — это популярная архитектура веб-приложений на PHP, которая позволяет разделять код на три основных компонента: модель, представление и контроллер.

Модель (Model) отвечает за работу с данными и обычно включает в себя классы для работы с базой данных, а также приложения для бизнес-логики.

Представление (View) отвечает за защиту данных и обычно включает в себя шаблоны HTML, CSS и JavaScript.

Контроллер (Controller) отвечает за обработку документов пользователей и управление взаимодействием между моделью и представлением.

Преимущества

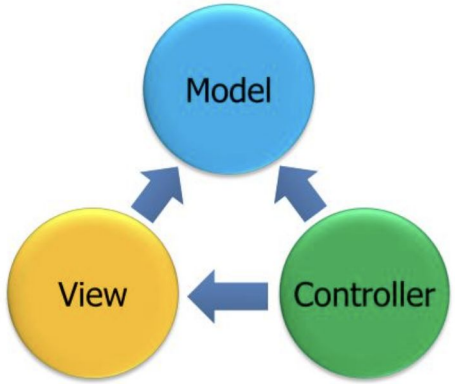
- Описание: код разделен на три компонента, что позволяет облегчить его сборку и поддержку.
- Гибкость: каждый компонент может быть изменен или заменен без описания других компонентов, что добавлено новых функций или изменений в приложении.
- Повторное использование кода: использование MVC-архитектур позволяет повторно использовать модели кода или представления в других частях приложений.

Недостатки

- Сложность: для начинающих может быть сложно разобраться в MVC-архитектуре и правильно ее реализовать.
- Накладные расходы на сборку: разделение кода на три компонента может привести к набору множества файлов и сложности разработки.

Вывод

Архитектура MVC на PHP является одним из наиболее распространенных подходов к разработке веб-приложений и высокой гибкостью повторного использования кода.



- **Model** (модель) — модель данных
- **View** (представление) – интерфейс
- **Controller** (контроллер)— логика

Шаблоны проектирования

Шаблоны разработки - это повторно используемые типичные решения, которые решают проблемы в проектах приложений. Они используют абстрактные модели, которые можно применять для решения конкретных задач.

В PHP шаблоны широко используются для упрощения разработки и улучшения качества кода. Они позволяют избежать ошибок, предусматривающих проектирование сложных систем, а также ускорение и расширение приложений.

Одним из преимуществ использования шаблонов проектирования в PHP является возможность создавать более гибкие и масштабируемые приложения. Шаблоны проектирования используются разработчиками, увеличивают производительность приложений, повышают его надежность и снижают затраты на разработку и поддержку.

Фабричный метод

Фабричный метод (Factory Method) — это шаблон проектирования, который предоставляет интерфейс для создания объектов в суперклассе, использует подклассы изменяемых типов используемых объектов.

Фабричный метод PHP широко используется для создания объектов в зависимости от условий. Он помогает разработчикам создавать объекты в более гибкой и расширяемой форме.

Реализация фабричного метода в PHP может выглядеть следующим образом:

```php
<?php
interface Animal {
    public function speak();
}

class Dog implements Animal {
    public function speak() {
        return "Woof!";
    }
}

class Cat implements Animal {
    public function speak() {
        return "Meow!";
    }
}

class AnimalFactory {
    public function createAnimal($type) {
        if ($type == "dog") {
            return new Dog();
        } else if ($type == "cat") {
            return new Cat();
        } else {
            throw new Exception("Invalid animal type.");
        }
    }
}
?>
```

Пояснения к коду:

В данном примере мы создали интерфейс Animal и два класса, реализующих этот интерфейс - Dog и Cat.

Затем мы разработали класс AnimalFactory, который содержит метод createAnimal, который в свою очередь принимает тип животного и подходит для животных.

Преимущества

- Гибкость: фабричный метод позволяет создавать объекты различных типов, что делает приложение более гибким и расширяемым.
- Улучшенная поддержка: фабричный метод поддержки получения приложений, так как он изолирует процесс создания объектов в одном месте.
- Увеличение производительности: фабричный метод может увеличить производительность приложений, так как создание объектов осуществляется в одном месте и не происходит в различных частях кода.

Недостатки

- Дополнительный код: фабричный метод требует написания дополнительного кода для реализации.
- Усложнение структуры: использование фабричного метода может усложнить структуру приложений, особенно если используются различные фабрики.

Метод «Одиночка»

Метод одиночка (Singleton) - это шаблон проектирования, который позволяет создать один экземпляр класса и реализовать глобальную точку доступа к этому экземпляру.

В методе реализации PHP одиночка может выглядеть следующим образом:

```php
<?php
class Singleton {
    private static $instance;

    private function __construct() {
        // Конструктор класса должен быть закрытым (private)
    }

    public static function getInstance() {
        if (!isset(self::$instance)) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }

    public function doSomething() {
        // Действия, которые должен выполнить объект класса
    }
}

?>
```

Пояснения к коду:

В данном примере мы создали класс Singleton, который содержит приватный реализованный метод $instance и открытый реализованный метод getInstance.

Метод getInstance создает только один экземпляр класса и использует глобальную точку доступа к этому экземпляру.

Метод doSomething представляет собой действие, которое должно контролировать объект класса.

Преимущества

- Глобальный доступ: глобальный доступ к единственному экземпляру класса.
- Улучшенная поддержка: метод одиночки получает поддержку приложения, так как он изолирует процесс создания объектов в одном месте.
- Потокобезопасность: метод одиночка собирается в многопоточных приложениях.

Недостатки:

- Затруднения при тестировании: тестирование класса Singleton может быть затруднено из-за глобального состояния.
- Ограничения на исследование: метод одиночка ограничивает возможность наследования класса и создания дополнительных экземпляров.

Метод «Стратегия»

Метод стратегии (Strategy) - это шаблон проектирования, который позволяет выбрать алгоритм во время выполнения программы. Он позволяет инкапсулировать различные алгоритмы в классах, которые можно заменять между собой без изменения кода, использующего эти алгоритмы.

В методе реализации PHP стратегия может выглядеть следующим образом:

```php
<?php
interface PaymentStrategy {
    public function pay($amount);
    }

class CreditCardStrategy implements PaymentStrategy {
    private $name;
    private $cardNumber;
    private $cvv;
    private $expirationDate;

    public function __construct($name, $cardNumber, $cvv, $expirationDate) {
        $this->name = $name;
        $this->cardNumber = $cardNumber;
        $this->cvv = $cvv;
        $this->expirationDate = $expirationDate;
    }

    public function pay($amount) {
       // Реализация оплаты кредитной картой
    }
}

class PayPalStrategy implements PaymentStrategy {
    private $email;
    private $password;

    public function __construct($email, $password) {
        $this->email = $email;
        $this->password = $password;
    }
```

```php
class PayPalStrategy implements PaymentStrategy {
    private $email;
    private $password;

    public function __construct($email, $password) {
        $this->email = $email;
        $this->password = $password;
    }

    public function pay($amount) {
       // Реализация оплаты через PayPal
    }
}

class Order {
    private $paymentStrategy;

    public function __construct(PaymentStrategy $paymentStrategy) {
        $this->paymentStrategy = $paymentStrategy;
    }

    public function processOrder($amount) {
        $this->paymentStrategy->pay($amount);
    }
}

?>
```

Пояснения к коду:

В данном примере мы создаем интерфейс PaymentStrategy, который содержит метод Pay.

Затем мы создаем два класса, реализующих этот интерфейс - CreditCardStrategy и PayPalStrategy. Эти классы инкапсулируют различные алгоритмы оплаты.

Класс Order использует интерфейс PaymentStrategy для выполнения платежей.

Преимущества

- Гибкость: метод стратегии позволяет легко изменять алгоритмы во время выполнения программы.
- Инкапсуляция: метод стратегии позволяет инкапсулировать различные алгоритмы в классе, которые вызывают реакцию и делают его более растворимым.
- Повторное использование кода: метод стратегии позволяет повторно использовать код, реализующий усовершенствованную архитектуру.

Недостатки

- Усложнение кода: метод стратегия может усложнить код приложения за счет создания дополнительных классов.
- Необходимость реализации: классы, реализующие метод стратегии, должны реализовывать защищенный интерфейс

Дополнения:

Применение метода стратегии в PHP особенно полезно в случае, когда необходимо учитывать какое-то действие с разнообразными вариантами поведения.

Например, веб-сайт магазина может иметь несколько вариантов оплаты: оплата кредитной картой, оплата через PayPal, оплата при получении и т.д. В этом случае можно создать отдельный класс для каждой желаемой оплаты, реализующий метод оплаты, который выполняет соответствующий алгоритм оплаты.

Класс заказа, который предлагает заказы, может использовать объекты классов для выполнения платежей.

Метод «Адаптер»

Метод адаптера в PHP является шаблоном проектирования, который позволяет объектам работать вместе с несовместимыми интерфейсами. Он используется в тех случаях, когда необходимо использовать имеющийся класс, но его интерфейс не соответствует требованиям востребованного кода.

Для реализации метода адаптера в PHP необходим адаптерный класс, который реализует требуемый интерфейс и содержит объект адаптируемого класса в качестве своего члена.

Метод «Адаптер»

Например, предположим, что у нас есть класс для работы с базой данных, который имеет методы для выполнения исходящих и получаемых результатов.

Однако для работы с другой базой данных нам нужен внешний интерфейс. Мы создаем адаптерный класс, который реализует требуемый интерфейс и содержит базу данных в качестве своего члена. В методах адаптерного класса используются методы данных.

Пример кода адаптера для работы с типами баз данных в PHP:

```php
<?php
interface DatabaseInterface {
    public function query($sql);
    public function getResult();
}

class MysqlDatabase {
    public function query($sql) {
        // выполнение запроса к базе данных MySQL
    }

    public function getResult() {
        // получение результата из базы данных MySQL
    }
}

class PostgresDatabase {
    public function select($query) {
        // выполнение запроса к базе данных Postgres
    }

    public function fetch() {
        // получение результата из базы данных Postgres
    }
}
```

```php
class MysqlAdapter implements DatabaseInterface {
    private $mysql;

    public function __construct(MysqlDatabase $mysql) {
        $this->mysql = $mysql;
    }

    public function query($sql) {
        $this->mysql->query($sql);
    }

    public function getResult() {
        $this->mysql->getResult();
    }
}

class PostgresAdapter implements DatabaseInterface {
    private $postgres;

    public function __construct(PostgresDatabase $postgres) {
        $this->postgres = $postgres;
    }

    public function query($sql) {
        $this->postgres->select($sql);
    }

    public function getResult() {
        $this->postgres->fetch();
    }
}
?>
```

Пояснение к коду:

В этом примере MysqlAdapter и PostgresAdapter являются адаптерными классами, реализующими интерфейс DatabaseInterface. Они являются объектами MysqlDatabase и PostgresDatabase, соответственно, и вызывают методы объектов в методах query и getResult. Это позволяет нам использовать один интерфейс для работы с типами баз данных.

Преимущества:

- Разделение интерфейсов: Адаптер позволяет разделять интерфейсы, что делает код более гибким и доступным.
- Поддержка старого кода: Адаптер позволяет использовать старый код в новых приложениях, что добавляет возможность «переиспользования» кода.
- Безопасность: обеспечение безопасности, например, для ограничения доступа к открытым данным.

Недостатки

- Усложнение кода: Адаптерный класс создает дополнительный уровень абстракции и усложняет код.
- Дополнительные затраты на производство: в нем есть дополнительные дополнительные ресурсы обработки.
- Необходимость разработки дополнительного кода: Для создания адаптерного класса требуется дополнительный код, что может увеличить затраты на разработку и тестирование.

Метод «Декоратор»

Метод декоратор (Decorator) в PHP позволяет добавлять новые функциональные возможности существующим объектом, не изменяя исходного кода. Декоратор является промежуточным получением, который оборачивает исходный объект и добавляет к нему свойство.

Рассмотрим пример декоратора для класса «Кофе» (Coffee):

```php
<?php
interface Coffee
{
    public function getCost(): float;
    public function getDescription(): string;
}

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

    public function getDescription(): string
    {
        return "Простой кофе";
    }
}

abstract class CoffeeDecorator implements Coffee
{
    protected $coffee;

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

```php
class Milk extends CoffeeDecorator
{
    public function getCost(): float
    {
        return $this->coffee->getCost() + 2;
    }

    public function getDescription(): string
    {
        return $this->coffee->getDescription() . ", молоко";
    }
}

class Sugar extends CoffeeDecorator
{
    public function getCost(): float
    {
        return $this->coffee->getCost() + 1;
    }

    public function getDescription(): string
    {
        return $this->coffee->getDescription() . ", сахар";
    }
}

?>
```

Пояснения к коду:

- В этом примере интерфейс Coffee определяет методы получения стоимости и описание кофе. Класс SimpleCoffee реализует этот внешний вид и представляет собой простой кофе.
- Абстрактный класс CoffeeDecorator дает базовую массу декораторов. Классы Milk и Sugar наследуются от CoffeeDecorator и добавляются к кофе молоко и сахар соответственно.

Преимущества:

- Расширяемость: Декоратор позволяет добавлять новую функциональность к объекту, не изменяя исходный код. Это делает декоратор гибким и расширяемым.
- Разделение ответственностей: Метод декоратор позволяет разделиться между классами, что делает код более читаемым и структурным для поддержки.
- Объектно-ориентированный подход: Декоратор использует объектно-ориентированный подход к программированию, что делает его более гибким и эффективным в сборке.
- Гибкость: Декоратор может включаться и удаляться в любое время во время выполнения программы, что позволяет создавать гибкие и реализуемые приложения.

Недостатки:

- Увеличение количества классов: Использование этого метода может отображать количество классов в приложении, что может усложнить его структуру и поддержку.
- Перегрузка объектов: Перегрузка объектов может привести к ухудшению производительности.
- Усложнение кода: Декоратор может усложнить код приложения, особенно если использовать цепочки декораторов, что может усложнить его чтение и понимание.

Шаблонный метод

Шаблонный метод - это установленный шаблон проектирования, который позволяет определить базовый алгоритм скелета в используемом классе и переопределить некоторые этапы этого алгоритма в дочерних классах без изменений общей структуры алгоритма.

В PHP шаблонный метод можно реализовать следующим образом:

```php
<?php
abstract class ReportGenerator {

    public function generateReport($data) {
        $this->createHeader();
        $this->generateContent($data);
        $this->createFooter();
    }
    protected function createHeader() {
        echo "Report Header\n";
    }
    protected abstract function generateContent($data);
    protected function createFooter() {
        echo "Report Footer\n";
    }
}

class PdfReportGenerator extends ReportGenerator {
    protected function generateContent($data) {
        echo "Generating PDF report content from data: $data\n";
    }
}

class CsvReportGenerator extends ReportGenerator {
    protected function generateContent($data) {
        echo "Generating CSV report content from data: $data\n";
    }
}

$pdfGenerator = new PdfReportGenerator();
$pdfGenerator->generateReport("PDF Report Data");
$csvGenerator = new CsvReportGenerator();
$csvGenerator->generateReport("CSV Report Data");
?>
```

Пояснения к коду:

- Этот класс ReportGenerator является абстрактным и содержит базовый алгоритм для генерации отчетов. Метод generateReport определяет общий порядок выполнения алгоритма, который включает создание заголовка, создание отчета и создание простого колонтитула. Однако метод generateContent генерации подклассов генерирует отчеты о реализации специфической логики генерации содержимого.
- Подклассы PdfReportGenerator и CsvReportGenerator реализуют метод generateContent, чтобы генерировать особое содержание в формате PDF и CSV соответственно. В этом примере используются два варианта отчета, один в формате PDF, другой в формате CSV, используется шаблонный метод для определения алгоритма генерации отчета.

Преимущества:

- Изменение поведения базового алгоритма без изменения его структуры. Подклассы могут переопределять только необходимые методы, не затрагивая базовый алгоритм.
- Разделение общего алгоритма на более мелкие методы, что включает понимание его кода и тестирование.
- Повторное использование кода. Базовый алгоритм может быть использован многими подклассами.

Недостатки:

- Ограничения на изменение базового алгоритма. При сборе шаблонного метода возможности изменения базового алгоритма ограничены, так как подклассы должны следовать общему порядку выполнения.
- Усложнение кода. Подклассы могут переопределять только определенные методы, что может привести к большому количеству подклассов.
- Отсутствие гибкости. Сборник шаблонного метода невозможного управления порядком выполнения базового алгоритма во время выполнения программы.

Заключение

Понимание архитектуры приложений и создание шаблонов на PHP является важным для создания масштабируемых и эффективных приложений.

Одним из основных преимуществ использования архитектурных шаблонов и проектирования шаблонов в PHP является то, что они упрощают разработку, обеспечивают поставку производства и облегчают сопровождение кода.

Мы рассмотрели несколько архитектурных паттернов и шаблонов проектирования на PHP, включая монолитную архитектуру, микросервисную архитектуру и шаблоны проектирования, такие как фабричный метод, метод одиночка, метод стратегии, метод адаптера и декоратор. Каждый из этих шаблонов проектирования имеет свои преимущества и недостатки, которые следует выбирать в наиболее подходящих условиях в конкретной ситуации.

Рефлексия

Сегодня на занятии мы познакомились с вами с архитектурой приложений и шаблонами проектирования. Давайте вспомним, чтобы мы сегодня узнали?

1) Какие шаблоны проектирования вы запомнили?
2) Какой шаблон проектирования вам показался наиболее интересным?
3) Что такое “архитектура приложения”?
4) Зачем нужна архитектура приложения?
5) Что вам больше всего понравилось на занятии?
6) Что было не понятно?
7) Чтобы вы хотели добавить в этот урок?

Список используемой литературы

1. «Шаблоны проектирования: элементы многоразового объектно-ориентированного программного обеспечения» Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона, Джона Влиссидеса.
2. «Чистая архитектура: руководство мастера по структуре и дизайну программного обеспечения» Роберта С. Мартина.
3. «Шаблоны архитектуры корпоративных приложений» Мартина Фаулера
4. «Объекты, шаблоны и практика PHP», Мэтт Зандстра
5. «Шаблоны проектирования Head First», Эрик Фриман, Элизабет Робсон, Берт Бейтс, Кэти Сьерра
6. «Освоение шаблонов проектирования PHP», Джунад Али
7. «Изучение шаблонов проектирования PHP», Уильям Сандерс, Келт Докинс
8. «Шаблоны проектирования PHP 7», Стив Преттимен


Спасибо
за
внимание