Приемы работы в ООП стиле

Шаблоны проектирования

Шаблоны проектирования или паттерны - это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Они впервые были представлены в книге Design Patterns (Erich Gamma, Richard Helm, Ralph Johnson и John Vlissides - известные как “банда четырёх”).
Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений.
По назначению паттерны делятся на порождающие, структурные и паттерны поведения:

Порождающие *(Creational)* — шаблоны проектирования, которые абстрагируют процесс создания. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять создаваемый класс, а шаблон, порождающий объекты, делегирует создание другому объекту.
Эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.А так же скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе - это их интерфейсы, определенные с помощью абстрактных классов. Поэтому, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда.
Структурные *(Structural)* определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.
Поведенческие *(Behavioral)* определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Одиночка (Singleton)

Одиночка (Singleton) относится к порождающим паттернам, гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Применение:

- *должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам*
- *единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода*

Например если нам не нужно много подключений к базе, можно воспользоваться данным паттерном:

```php
class Db
{
    private $_conn;
    private static $_instance;  // экземпляр объекта
    // Защищаем от создания через new
    private function __construct()
    {
        $dsn = 'mysql:host=localhost;dbname=db_name;charset=utf8';
        $this->_conn = new PDO($dsn, 'user', 'password');
    }
    // Защищаем от создания через клонирование
    private function __clone() {}
    // Защищаем от создания через unserialize
    private function __wakeup() {}

    // Возвращает единственный экземпляр класса Db
    public static function getInstance()
    {
        if (empty(self::$_instance))
        {
            self::$_instance = new self();
        }
        return self::$_instance;
    }

    public function querySql($sql)
    {
        $result = $this->_conn->query($sql);
        return $result->fetchAll(PDO::FETCH_ASSOC);
    }

}

// new Db(); - ОШИБКА __construct() private
Db::getInstance()->querySql('SELECT...');
```

Фабрика (Factory)

Фабрика (Factory) относится к порождающим паттернам, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не определяя их конкретных классов. Например если заранее не известно объекты каких классов создавать, или хотим скрыть логику создания сложного объекта или же хотим уменьшить связи между объектами. Существует несколько реализаций данного паттерна, но суть схожа:

- ***Абстрактная фабрика (Abstract Factory)*** - класс, который представляет собой интерфейс для создания компонентов системы.
- ***Фабричный метод (Factory Method)*** - определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Применение:

- *классу заранее неизвестно, объекты каких классов ему нужно создавать*
- *класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами*
- *класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя*

Рассмотрим Фабричный метод. Создадим базовый абстрактный класс ABaseParser парсера, у которого есть абстрактный метод read()

```php
abstract class ABaseParser
{
    protected $file;

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

    abstract function read();
}
```

Теперь создадим классы наследники которые будут реализовывать этот метод.
XmlParser для xml файлов

```php
class XmlParser extends ABaseParser
{
    public function read()
    {
        echo 'Здесь будет реализация чтения xml файла<br>';
    }
}
```

JsonParser для json файлов

```php
class JsonParser extends ABaseParser
{
    public function read()
    {
        echo 'Здесь будет реализация чтения json файла<br>';
    }
}
```

Эти классы реализуют метод read каждый по своему. Теперь представьте что мы хотим обработать множество файлов, для каждого файла пришлось бы писать создание конкретного класса, что очень плохо. Но если мы создадим дополнительный класс который сам будет решать какой класс создавать, это решит нашу проблему.

```php
class ParserFactory
{
    public static function create($file)
    {
        $data = pathinfo($file);
        $clsname = ucfirst($data['extension']) .'Parser';

        if (class_exists($clsname) && is_subclass_of($clsname, 'ABaseParser'))
        {
            return new $clsname($file);
        }

        throw new Exception('Некорректный файл');
    }
}
```

Теперь мы можем обрабатывать множество файлов очень удобным способом:

```php
$files = ['data.xml', 'data.json'];

foreach ($files as $file)
{
    $parser = ParserFactory::create($file);
    $parser->read();
}
```

Стратегия (англ. Strategy)

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

Применение:

- *имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений*
- *вам нужно иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой - больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов*
- *в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных*
- *в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.*

К примеру в нашем интернет магазине при оплате суммы до 5000 используется робокасса, иначе карта. Чтобы не вставлять проверку этого условия при каждой оплате перенесем её в класс Shop, и создадим отдельные классы для оплаты через робокассу и для оплаты картой, которые будут реализовывать интерфейс I_Payment

```php
interface I_Payment
{
    public function pay($sum);
}
```

```php
class RobokassaPay implements I_Payment
{
    public function pay($sum)
    {
        echo 'Оплата через Robokassa суммы: '. $sum;
    }
}
```

```php
class CardPay implements I_Payment
{
    public function pay($sum)
    {
        echo 'Оплата картой суммы: '. $sum;
    }
}
```

```php
class Shop
{
    public function payForPurchases($sum)
    {
        if ($sum >= 5000) // Если больше 5000 то картой
        {
            $payment = new CardPay();
        }
        else // Иначе оплата картой
        {
            $payment = new RobokassaPay();
        }

        $payment->pay($sum);
    }
}
```

```php
$shop = new Shop();
$shop->payForPurchases(5000); // Оплата картой суммы: 5000
$shop->payForPurchases(1000); // Оплата через Robokassa суммы: 1000
```

Наблюдатель (Observer)

Наблюдатель (Observer) паттерн поведения, определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Применение:

- *когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо*
- *когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить*
- *когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой*

Примером реализации может быть подписка пользователей на рассылку. Создадим интерфейс для “подписчика”

```
interface IObserver
{
    public function notify(IObservable $obj, $title);
}
```

А так же интерфейс для “издателя” который будет подписывать и уведомлять, он так же может удалять подписчиков но нам это не понадобится.

```
interface IObservable
{
    public function addObserver(IObserver $obj);

    public function notifyObservers();
}
```

Создадим класс издателя реализовав соответствующий интерфейс

```php
class DeliveryEmails implements IObservable
{
    private $_title;
    private $_observers = [];
    private static $_instance;

    public static function getInstance()
    {
        if (empty(self::$_instance))
        {
            self::$_instance = new self();
        }
        return self::$_instance;
    }
    // Добавляем новую статью для уведомления
    public function addNewArticle($title)
    {
        $this->_title = $title;
        $this->notifyObservers();
    }
    // Добавляем подписчиков
    public function addObserver(IObserver $obj)
    {
        $this->_observers[] = $obj;
    }
    // Уведомляем подписчиков
    public function notifyObservers()
    {
        foreach ($this->_observers as $observer)
        {
            $observer->notify($this, $this->_title);
        }
    }

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

Теперь класс пользователя реализующий интерфейс подписчика

```php
class User implements IObserver
{
    private $_email;

    public function __construct($email)
    {
        $this->_email = $email;
        // Подписываем данного пользователя на уведомление
        DeliveryEmails::getInstance()->addObserver($this);
    }
    // Уведомление пользователя
    public function notify(IObservable $obj, $title)
    {
        if ($obj instanceof IObservable)
        {
            echo "$title отправлено на {$this->_email}<br>";
        }
    }
}
```

Создадим пользователей и добавим новую статью

```php
$vasya = new User('Vasya@perchik.ru');
$masha = new User('Masha@netkrashe.ru');
DeliveryEmails::getInstance()->addNewArticle('Свежие новости');
```

Адаптер (Adapter)

Адаптер (Adapter) структурный шаблон проектирования, при котором объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.

Применение:

- *хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям*
- *собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы*
- *(только для адаптера объектов!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса*

Представьте что мы работаем с api к примеру vk. В один прекрасный момент изменилось название метода класса этого api отвечающего за размещение записи на стене и нам придется вносить изменения везде где мы его использовали. Чтобы этого избежать можно воспользоваться данным паттерном:

```php
// Интерфейс адаптера
interface I_VkAdapter
{
    public function postOnWall($text);
}
```

```php
//Старое Api
class VkApi1
{
    public function send($text)
    {
        echo 'Old: '. $text;
    }
}
```

```php
//Новое Api
class VkApiNew
{
    public function sendPost($text)
    {
        echo 'New: '. $text;
    }
}
```

```php
// Класс адаптера
class VkAdapter implements I_VkAdapter
{
    private $_vk_api;

    public function __construct($vk_api)
    {
        $this->_vk_api = $vk_api;
    }
    // При изменении меняется только этот метод
    public function postOnWall($text)
    {
        $this->_vk_api->sendPost($text);
    }
}
```

Теперь все изменения производятся в одном месте а именно в VkAdapter, тогда как остальной код и код api остается без изменений.

```php
$vk_adapter = new VkAdapter(new VkApiNew());
$vk_adapter->postOnWall('Привет VK!'); // New: Привет VK!
```