Обучение Советы

Шаблоны проектирования по-человечески: поведенческие паттерны в примерах

От **Montgomeri** - 09.06.2017

Поведенческие паттерны отвечают за эффективное взаимодействие объектов. В отличие от структурных, они также затрагивают шаблоны для обмена сообщениями.

Главной проблемой чтения чужого кода могут быть паттерны. Но даже если вы работаете с исконно своим проектом, наша подборка облегчит вашу жизнь. Возьмите на карандаш 😉

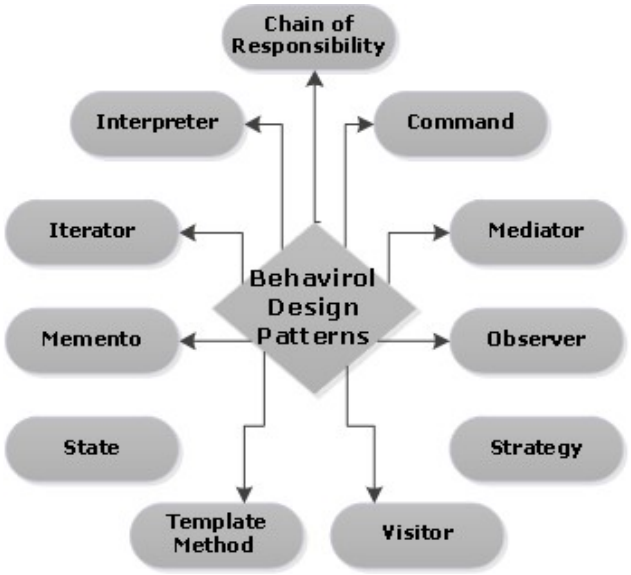
Характеристика

Данные шаблоны проектирования определяют способы и алгоритмы реализации совместной работы различных классов и объектов. Поведенческие паттерны уровня класса используют механизм наследования, а шаблоны уровня объекта – механизм композиции. Соседствующие объекты «знают» друг о друге, и важно то, как именно реализована эта связь.

Поведенческие паттерны: классификация

Они делятся на:

1. Chain of Responsibility
2. Command
3. Iterator
4. Mediator
5. Memento
6. Observer
7. Visitor
8. Strategy
9. State
10. Template Method



1. Паттерн Chain of Responsibility

Предположим, вы хотите оплатить покупку в Интернете. Всего есть 3 способа оплаты: A, B и C. Каждый из них располагает суммами в $100, $300 и $1000 соответственно, а приоритетность способов снижается от A к C. Вы хотите приобрести товар стоимость $210. Сперва будет проверен баланс A. Если нет достаточной суммы, запрос перемещается к балансу B, и так далее, пока не будет найдена необходимая сумма. Именно так работают поведенческие паттерны Цепочка Обязанностей, где A, B и C – звенья.

Итак, у нас есть базовая учетная запись, в которой объединены дополнительные учетные записи:

```
1  abstract class Account
2  {
3      protected $successor;
4      protected $balance;
5
6      public function setNext(Account $account)
7      {
8          $this->successor = $account;
9      }
10
```

```
    public function pay(float $amountToPay)
    {
        if ($this->canPay($amountToPay)) {
            echo sprintf('Оплачено %s с использованием %s' . PHP_EOL, $amountToPay, get_called_class()
        } elseif ($this->successor) {
            echo sprintf('Нельзя оплатить с использованием %s. В процессе ..' . PHP_EOL, get_called_cl
            $this->successor->pay($amountToPay);
        } else {
            throw new Exception('Ни на одном из аккаунтов нет необходимых средств');
        }
    }

    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 просто подготовим цепь, используя банковский счет, Paypal и Bitcoin:

```
$bank = new Bank(100);          // Банковский счет 100
$paypal = new Paypal(200);      // Paypal 200
$bitcoin = new Bitcoin(300);    // Bitcoin 300

$bank->setNext($paypal);
$paypal->setNext($bitcoin);

// Попробуем оплатить с помощью приоритетного банковского счета
$bank->pay(259);

// Выдаст:
// ==============
// Нельзя оплатить с использованием bank. В процессе ..
// Нельзя оплатить с использованием paypal. В процессе ..
// Оплачено 259 с использованием Bitcoin
```

2. Паттерн Команда

Заказ еды в ресторане: вы (Client) попросите официанта (Invoker) принести еду (Command), а официант отправит этот запрос шеф-повару (Receiver). Другой пример для паттерна Command: вы (Client)

включаете (Command) телевизор (Receiver) с помощью пульта (Invoker).

Прежде всего, у нас есть получатель (Receiver), который выполняет определенные действия:

```php
// Receiver
class Bulb
{
    public function turnOn()
    {
        echo "Лампа светится";
    }

    public function turnOff()
    {
        echo "Тьма";
    }
}
```

Есть интерфейс реализации команд и набор команд:

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

// Command
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();
    }
```

```
52
53      public function redo()
54      {
55          $this->execute();
56      }
57  }
```

Также должен быть посредник (Invoker), с которым взаимодействует клиент:

```
1  // Invoker
2  class RemoteControl
3  {
4      public function submit(Command $command)
5      {
6          $command->execute();
7      }
8  }
```

А теперь смотрим, как это все работает:

```
1  $bulb = new Bulb();
2
3  $turnOn = new TurnOn($bulb);
4  $turnOff = new TurnOff($bulb);
5
6  $remote = new RemoteControl();
7  $remote->submit($turnOn); // Лампа светится
8  $remote->submit($turnOff); // Тьма
```

3. Паттерн Итератор

Хорошим примером станет старый радиоприемник, в котором можно переключаться на следующий и предыдущий каналы с помощью соответствующих кнопок. Нечто подобное можно проделать с телевизором, MP3-плеером и прочими устройствами. Именно поведенческие паттерны Iterator позволяют обходить элементы объекта последовательно.

С PHP это легко реализовать, используя SPL (Standard PHP Library). Разберем пример. Изначально у нас есть радиостанция:

```
1  class RadioStation
2  {
3      protected $frequency;
4
5      public function __construct(float $frequency)
6      {
7          $this->frequency = $frequency;
8      }
9
10     public function getFrequency(): float
11     {
12         return $this->frequency;
13     }
14 }
```

Далее вводим итератор:

```
1  use Countable;
2  use Iterator;
3
4  class StationList implements Countable, Iterator
5  {
6      /** @var RadioStation[] $stations */
7      protected $stations = [];
```

```php

    /** @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 $station) use ($toRemov
            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]);
    }
}
```

Реализовываем:

```php
$stationList = new StationList();

$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)); // Возвращает станцию 89
```

4. Mediator паттерн

Когда вы разговариваете с кем-то по телефону, это никогда не происходит напрямую. Между вами и собеседником находится провайдер, и в этом случае поставщик мобильных услуг является посредником.

Рассмотрим на примере чата, где и будут использованы поведенческие паттерны Посредник. Есть окно чата:

```php
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;
    }
}
```

Добавляем к нему пользователей:

```php
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);
    }
}
```

Реализовываем:

```php
$mediator = new ChatRoom();

$john = new User('Джон', $mediator);
$jane = new User('Джейн', $mediator);

$john->send('Здравствуй!');
$jane->send('Привет!');

// Output will be
// Feb 14, 10:58 [Джон]: Здравствуй!
// Feb 14, 10:58 [Джейн]: Привет!
```

5. Memento паттерн

Когда вы делаете какой-либо рассчет с помощью калькулятора, последнее действие сохраняется в памяти устройства. Это нужно для того, чтобы к этому действию можно было вернуться и, возможно, восстановить, нажав на определенные кнопки.

Поведенческие паттерны Memento можно рассмотреть и на примере текстового редактора, который периодически делает сохранения. У нас есть объект, который сможет удерживать состояние редактора:

```php
class EditorMemento
{
    protected $content;

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

    public function getContent()
    {
        return $this->content;
    }
}
```

Появляется сам редактор:

```php
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();
    }
}
```

Используем:

```php
$editor = new Editor();

// Ввод
$editor->type('Это первое предложение.');
$editor->type('Это второе.');

// Сохраняем состояние для восстановления: Это первое предложение. Это второе.
$saved = $editor->save();

// Вводим еще
$editor->type('И это уже третье.');

// Вывод: содержание перед сохранением
echo $editor->getContent(); // Это первое предложение. Это второе. И это уже третье.

// Восстановление последнего сохраненного состояния
$editor->restore($saved);

$editor->getContent(); // Это первое предложение. Это второе.
```

6. Паттерн Наблюдатель

Люди, которые ищут работу, часто подписываются на сайты, где публикуются вакансии. Именно эти сайты уведомляют соискателей о подходящих должностях, и именно так работают поведенческие

паттерны Observer.

Есть соискатели:

```php
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)
    {
        // Do something with the job posting
        echo 'Привет ' . $this->name . '! Размещена новая вакансия: '. $job->getTitle();
    }
}
```

Добавляем вакансии, на которые можно подписываться:

```php
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);
    }
}
```

Используем:

```php
$johnDoe = new JobSeeker('Джон');
$janeDoe = new JobSeeker('Джейн');

$jobPostings = new JobPostings();
$jobPostings->attach($johnDoe);
$jobPostings->attach($janeDoe);

$jobPostings->addJob(new JobPost('Разработчик ПО'));
```

7. Паттерн Visitor

Предположим, вы решили посетить Дубай. Для этого понадобится только виза. По прибытии вы можете посетить любое место города самостоятельно, без необходимости получать дополнительные разрешения. Просто узнайте о нужном месте и посетите его. Поведенческие паттерны Посетитель как раз и отвечают за добавление таких мест, которые можно посещать без дополнительных утруждающих действий.

В качестве примера для кода возьмем зоопарк, где есть разные животные. Зададим интерфейс:

```php
interface Animal
{
    public function accept(AnimalOperation $operation);
}

interface AnimalOperation
{
    public function visitMonkey(Monkey $monkey);
    public function visitLion(Lion $lion);
    public function visitDolphin(Dolphin $dolphin);
}
```

Работаем с разными видами животных:

```php
class Monkey implements Animal
{
    public function shout()
    {
        echo 'Ooh oo aa aa!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitMonkey($this);
    }
}

class Lion implements Animal
{
    public function roar()
    {
        echo 'Roaaar!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitLion($this);
    }
}

class Dolphin implements Animal
{
    public function speak()
    {
        echo 'Tuut tuttu tuutt!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitDolphin($this);
    }
}
```

Реализуем посетителя:

```php
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();
    }
}
```

Используем:

```php
$monkey = new Monkey();
$lion = new Lion();
$dolphin = new Dolphin();

$speak = new Speak();

$monkey->accept($speak);     // Ooh oo aa aa!
$lion->accept($speak);       // Roaaar!
$dolphin->accept($speak);    // Tuut tutt tuutt!
```

8. Паттерн Стратегия

Рассмотрим пример сортировки пузырьком. Когда данных становится слишком много, такой вид сортировки становится очень медленным. Чтобы решить проблему, мы применим быструю сортировку. Но хоть этот алгоритм и обрабатывает большие объемы быстро, в небольших он медленный. Поведенческие паттерны Strategy позволяют реализовать стратегию, в которой совмещены оба метода.

Имеем интерфейс и различные варианты реализации стратегии:

```php
interface SortStrategy
{
    public function sort(array $dataset): array;
}

class BubbleSortStrategy implements SortStrategy
{
    public function sort(array $dataset): array
    {
        echo "Сортировка пузырьком";

        // Сортируем
        return $dataset;
    }
}

class QuickSortStrategy implements SortStrategy
{
    public function sort(array $dataset): array
    {
        echo "Быстрая сортировка";

        // Сортируем
        return $dataset;
    }
}
```

Теперь есть клиент, выбирающий один из вариантов:

```
class Sorter
{
    protected $sorter;

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

    public function sort(array $dataset): array
    {
        return $this->sorter->sort($dataset);
    }
}
```

Используем:

```
$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
```

9. Паттерн Состояние

Допустим, вы используете приложение для рисования, где выбираете кисть. Теперь кисть меняет свое состояние в соответствии с выбранным цветом. То есть, если вы выбрали красный цвет, то и кисть будет рисовать красным.

Для кода используем пример с текстовым редактором, в котором можно изменить шрифт. У нас есть интерфейс и реализация некоторых состояний:

```
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;
    }
}
```

Добавляем редактор:

```php
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);
    }
}
```

Реализовываем:

```php
$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');

// Вывод:
// Первая строка
// ВТОРАЯ СТРОКА
// ТРЕТЬЯ СТРОКА
// четвертая строка
// пятая строка
```

10. Паттерн Шаблонный Метод

Строим дом. Этапы работы выглядят так:

- подготовка фундамента;
- строительство стен;
- добавление крыши;
- добавление необходимого количества этажей.

Порядок неизменен, но можно изменить каждый из этапов отдельно. Например, стены могут быть из камня или из дерева.

Предположим, есть инструмент для сборки, который позволяет тестировать программу, анализировать, генерировать отчеты и т. д. Создадим базовый класс-скелет:

```php
abstract class Builder
{
```

```
3
4       // Шаблонный Метод
5       final public function build()
6       {
7           $this->test();
8           $this->lint();
9           $this->assemble();
10          $this->deploy();
11      }
12
13      abstract public function test();
14      abstract public function lint();
15      abstract public function assemble();
16      abstract public function deploy();
17  }
```

Теперь реализации:

```
1   class AndroidBuilder extends Builder
2   {
3       public function test()
4       {
5           echo 'Старт Android тестов';
6       }
7
8       public function lint()
9       {
10          echo 'Анализ Android кода';
11      }
12
13      public function assemble()
14      {
15          echo 'Сборка Android';
16      }
17
18      public function deploy()
19      {
20          echo 'Развертывание Android';
21      }
22  }
23
24  class IosBuilder extends Builder
25  {
26      public function test()
27      {
28          echo 'Старт iOS тестов';
29      }
30
31      public function lint()
32      {
33          echo 'Анализ iOS кода';
34      }
35
36      public function assemble()
37      {
38          echo 'Сборка iOS';
39      }
40
41      public function deploy()
42      {
43          echo 'Развертывание iOS';
44      }
45  }
```

Используем:

```
1   $androidBuilder = new AndroidBuilder();
2   $androidBuilder->build();
3
4   $iosBuilder = new IosBuilder();
5   $iosBuilder->build();
```

Также рекомендуем Вам посмотреть:

- Шаблоны проектирования по-человечески: 6 порождающих паттернов, которые упростят жизнь
- Шаблоны проектирования по-человечески: структурные паттерны
- Лучший видеокурс по шаблонам проектирования
- 4 лучших книг о шаблонах проектирования
- 20 полезных навыков, которые можно освоить за 3 дня

Теги **Разное**

Предыдущая статья

164 крутых опенсорс проекта для новичков

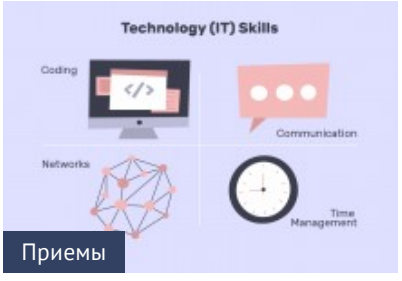
Следующая статья

Путь Python Junior-а в 2017

Похожие статьи | Больше от автора

Приемы

Истории



Приемы

Как перестать писать плохой код на Bash: практические советы

ТОП-5 историй о том, как IT-консалтинг ускоряет бизнес

10 советов: как подтянуть разговорный английский язык

Комментариев: 4

DenQ 19.09.2017, в 20:05

0

Как можно рассказывать о паттернах и не применять UML.

Ответить

Комментарий:

Добавить

Свежие вакансии

OpenGL ES. Удаленно, на долгосрочную перспективу.

ОТКЛИКНУТЬСЯ НА ВАКАНСИЮ

Plarium Krasnodar

Видеограф (Videographer)

Краснодар | полный день

Специалист отдела Video Production, отвечающий за съемку видео и финальную сборку воедино всех элементов медиаконтента. Создание итогового видео, состоящего из моушн-графики, фотоконтента, графических элементов, музыки, дикторской начитки.

ОТКЛИКНУТЬСЯ НА ВАКАНСИЮ

Темы

Android Blockchain C# C++ Data Science Frontend Go Hacking iOS Java JavaScript Junior Linux Middle Mobile Python Senior Web Алгоритмы Базы данных Математика Новичку Общее Разное Разработка игр Советы Трудоустройство

Случайные статьи

Собеседование для Data Scientists: вопросы и ответы

Как получить работу мечты

3 главные ошибки, которые вредят производительности JavaScript

Схема успешного развития data-scientist специалиста в 2019 году

Большая подборка книг, видео и статей для Java Middle

Популярные материалы

ТОП-8 трендов web-разработки, обязательных в 2019 году

14 советов, с которыми ты начнёшь мыслить как программист

ТОП-13 крутых идей веб-проектов для прокачки навыков

Хороший, спорный, злой Vue.js: опыт перехода с React

Твиттер на Vue.js: руководство для начинающих

Загрузить больше

О нас

Библиотека программиста — ваш источник образовательного контента в IT-сфере. Мы публикуем обзоры книг, видеолекции и видеоуроки, дайджесты и образовательные статьи, которые помогут вам улучшить процесс познания в разработке.

Подпишись

ВКонтакте | Telegram | Facebook | Instagram | Яндекс.Дзен

Медиакит | Пользовательское соглашение | Политика конфиденциальности

Связаться с нами

По вопросам рекламы: matvey@proglib.io

Для обратной связи: hello@proglib.io
123022, Москва, Рочдельская ул., 15, к. 17-18, +7 (995) 114-98-90