بسم الله الرحمن الرحیم

الگوهای طراحی به بیان ساده

مؤلف/مترجم:
احسان رضایی

NAGHOOS PUBLICATION

1399

سرشناسه	: رضایی،احسان،۱۳۷۱-
عنوان و نام پدیدآور	: الگوهای طراحی به بیان ساده / مولف احسان رضایی
مشخصات نشر	: تهران: انتشارات ناقوس، ۱۳۹۹.
مشخصات ظاهری	: ۱۰۴ص.: مصور.
شابک	: ۹-۲۷۸-۴۷۳-۶۰۰-۹۷۸ ۴۵۰,۰۰۰ریال
وضعیت فهرستنویسی	: فیپا
موضوع	: الگوهای نرم افزاری
موضوع	: Software patterns
موضوع	: نرم افزار مهندسی
موضوع	: Softwarte engineering
رده بندی کنگره	: QA۷۶/۷۶
رده بندی دیویی	: ۰۰۵/۱
شمارهٔ کتابشناسی ملی	: ۶۱۳۴۳۹۳

برای خرید Online به آدرس زیر مراجعه کنید:

developit.ir/design-patterns

NAGHOOS PUBLICATION

انتشارات ناقوس

نام کتاب	: الگوهای طراحی به بیان ساده
ترجمه و تالیف	: احسان رضایی
ناشر	: انتشارات ناقوس
چاپ اول	: ۱۳۹۹
تیراژ	: ۱۰۰ نسخه
چاپ و صحافی	: برجسته
قیمت	:۴۵۰,۰۰۰ریال
شابک دوره	: ۹-۲۷۸-۴۷۳-۶۰۰-۹۷۸
ISBN	: 978-600-473-278-9

کلیه حقوق برای سرمایه گذار محفوظ است. تکثیر تمامی یا قسمتی از این اثر به‌صورت حروفچینی یا چاپ مجدد، چاپ افست،

پلی‌کپی، فتوکپی و انواع دیگر چاپ ممنوع است و پیگرد قانونی دارد.

درباره مؤلف/مترجم

احسان رضایی

متولد سال 1371. از 16 سالگی برنامه نویسی تحت وب با php رو شروع کردم، به عنوان زبان برنامه نویسی دوم و رفتن به سمت تجربه‌های جدید golang رو یاد گرفتم.
نتیجه این تجربه جدید جاب ترندز بود. ابزاری برای تجزیه و تحلیل هوشمند بازار کار ایران.
http://job-trends.developit.ir

به علت اینکه در شهر خودم (قم) شغل مناسب پیدا نکردم و مخالف مهاجرت به شهرهای دیگه مثل تهران بودم دورکار شدم :)

در این لحظه که مشغول تکمیل بخش‌های نهایی کتابم، عضوی از تیم backend شرکت 3rdly هستم. داخل ایران هم توسعه یک بستر فریلنسری در زمینه تخصصی تایپ رو انجام میدم.

میتونید اخبار و بروزرسانی‌های جدید کتاب رو داخل وبلاگ یا توییترم دنبال کنید.

page: http://www.developit.ir/design-patterns
blog: http://www.developit.ir
email: me@developit.ir
twitter: @developit121

سخن مؤلف/مترجم

منابع زیادی برای مطالعه دیزاین پترن همراه با مثال‌هایی از زبان‌های مختلف برنامه نویسی وجود داره. اما همونطور که از نام این کتاب پیداست هدف بیان ساده الگوهای طراحیه. من فکر می‌کنم اگر مفاهیم رو به سادگی درک کنید ادامه مسیر هم براتون آسون خواهد بود. چیزی که ثابت میکنه شما به مهارت خوبی رسیدین تمرین هست و من تصمیم دارم در این مسیر مفاهیم رو به سادگی بهتون منتقل کنم.

برای گردآوری این کتاب منابع زیادی رو مطالعه/ ترجمه کردم مثل refactoring.guru، sourcemaking.com، کتاب Design Patterns: Elements of Reusable Object-Oriented Software و Design Patterns for Humans و نهایتاً کمی تجارب خودم.

بین این موارد از ساختار و روش Design Patterns for Humans که نزدیک‌ترین به هدف من بود و مجوز هاش محدودم نمی‌کرد خوشم اومد و با همین ساختار جلو رفتم.

در این کتاب از چه زبان برنامه نویسی استفاده شده؟

نگران نباشید! همونطور که با مطالعه کتاب متوجه خواهید شد اولویت درک مفاهیم و پیاده سازی الگوهای طراحی است. در نتیجه زبان برنامه نویسی در اینجا اهمیت اول رو نداره.

اما میتونید برای تمرین و مشاهده مثال‌های بیشتر پس از مطالعه هر بخش از کتاب به لینک زیر مراجعه کنید.

در این آدرس مثال‌های زیادی از الگوهای طراحی با زبان‌های مختلف مثل **جاوا، پایتون، سی پلاس پلاس، سی شارپ و...** وجود داره.

https://github.com/RefactoringGuru

آوردن مثال از چند زبان برنامه نویسی باعث افزایش حجم کتاب می‌شد و من این موضوع رو نمی‌پسندیدم. به هر حال داخل کتاب مثال‌هایی باید مطرح می‌کردم و من به عنوان نمونه اونها رو با PHP نوشتم.

نکته آخر اینکه به علت ساده بودن مثال‌ها ممکن هست در بخش‌هایی، یک مثال علاوه بر الگوی مورد نظر، سایر الگوهای طراحی رو هم پوشش بده.

فهرست مطالب

فصل اول: مقدمه

مقدمه 3
الگوی طراحی چیست؟ 3
چرا باید الگوهای طراحی را آموخت؟ 4
انواع الگوهای طراحی 4

فصل دوم: الگوهای طراحی سازنده / Creational

Simple Factory 9
Factory Method 12
Abstract Factory 15
Builder 19
Prototype 22
Singleton 25

فصل سوم: الگوهای طراحی ساختاری / Structural

Adapter 31
Bridge 34
Composite 38
Decorator 44
Facade 46
Flyweight 49
Proxy 52

فصل چهارم: الگوهای طراحی رفتاری / Behavioral

Chain of Responsibility 59
Command 63
Iterator 67

71	Mediator
74	Memento
76	Observer
80	Visitor
85	Strategy
88	State
81	Template Method

فصل اول

مقدمه

مقدمه

در مهندسی نرم‌افزار، design patterns (الگوهای طراحی) راه حل‌های قابل استفاده برای مشکلاتی هستند که معمولاً در طراحی نرم‌افزار اتفاق می‌افتند.
آن‌ها راه حل‌هایی برای عبور از مشکلات هستند. دستورالعمل‌هایی در مورد چگونگی مقابله با مشکلات. الگوهای طراحی کلاس‌ها، بسته‌ها و یا کتابخانه‌هایی که بتوانید به نرم افزار خود اضافه کنید و منتظر یک جادو باشید نیستند. همچنین طرح‌های نهایی که بتوانید آن‌ها را مستقیماً به کد تبدیل کنید نخواهند بود. الگوهای طراحی فقط چگونگی حل مسأله را شرح می‌دهند.
این الگوها می‌توانند سرعت توسعه و طول عمر نرم‌افزار را افزایش دهند. علاوه بر این، با توجه به تعاریف مشخص و شناخته شده‌ای که دارند هنگام توسعه نرم‌افزار در پروژه‌های تیمی به توسعه دهندگان قابلیت تعامل و برقراری ارتباط (بدون زجر) را خواهند داد.
مراقب باشید!
- الگوهای طراحی یک راه حل برای تمام مشکلات شما نیستند.
- به خاطر داشته باشید الگوهای طراحی راه حل‌هایی برای چالش‌ها و مشکلات هستند، نه راه حل‌هایی برای یافتن مشکلات. بنابراین زیاده روی نکنید.
- اگر در محل درست به صورت صحیح استفاده شوند. آن‌ها می‌توانند ثابت کنند که یک ناجی هستند در غیر این صورت می‌توانند منجر به آشفتگی در کد شوند.

الگوی طراحی چیست؟

همانطور که در مقدمه گفته شد راه حل‌های قابل استفاده برای مشکلاتی هستند که معمولاً در طراحی نرم‌افزار اتفاق می افتند.
طرح‌های از پیش ساخته شده‌ای که می‌توانید برای حل مشکلات آن‌ها را سفارشی کنید. شما نمی‌توانید یک الگو را با جستجو در stackoverflow پیدا و در برنامه خود

کپی کنید. **الگوها یک قطعه کد خاص نیستند، مفاهیم کلی برای حل مشکلات خاص هستند.** شما باید با درک این مفاهیم آن‌ها را در برنامه خود پیاده‌سازی کنید.

چرا باید الگوهای طراحی را آموخت؟

ممکن است سال‌ها به عنوان یک برنامه نویس بدون استفاده از الگوهای طراحی کار کرده باشید. بسیاری این کار را انجام می‌دهند! شاید بعد از مطالعه کتاب متوجه شوید برخی از الگوها را بدون دانستن آن‌ها استفاده کرده‌اید. پس چرا باید الگوهای طراحی را یاد گرفت؟

الگوهای طراحی راه حل‌های قابل استفاده برای مشکلاتی هستند که در طراحی نرم‌افزار اتفاق می افتند. حتی اگر هرگز با این مشکلات روبرو نشده‌اید که امکان ندارد! شناختن آن‌ها هنوز هم مفید است زیرا به شما می‌آموزد چگونه این مشکلات حل شدنی هستند.

الگوهای طراحی یک زبان مشترک تعریف می‌کنند. که شما و سایر اعضای تیم می‌توانید برای برقرای ارتباط مؤثر از آن استفاده کنید.

من در پروژه جاب ترندز مشغول تجزیه و تحلیل بازار کار هستم. در بیش از 65 درصد آگهی‌های استخدام برنامه نویس کشور، نیاز به دانش و تخصص الگوهای طراحی مطرح شده. در نتیجه آموختن این الگوها در آینده شغلی شما تأثیر مثبتی خواهد گذاشت.

http://job-trends.developit.ir

انواع الگوهای طراحی

بر اساس کتاب Design Patterns: Elements of Reusable Object-Oriented Software سه نوع الگوی طراحی وجود دارد.

- creational / سازنده

مکانیسم‌های ایجاد اشیاء را فراهم می‌کنند و انعطاف پذیری و استفاده مجدد را افزایش می‌دهند.

- structural / ساختاری

به جمع‌آوری اشیاء و کلاس‌های مرتبط در ساختارهای بزرگ و منسجم‌تر می‌پردازند.

- behavioral / رفتاری

ارتباط مؤثر بین اشیاء و اختصاص وظایف را بهبود می‌بخشند.

فصل دوم

الگوهای طراحی سازنده / Creational

الگوهای طراحی سازنده / Creational

تعریف instantiation

به ایجاد یک نمونه از یک شئ/کلاس گفته می‌شود.

```
$manager = new Manager();
```

به بیان ساده

الگوهای creational روی ایجاد یک شئ یا گروهی از اشیاء مرتبط تمرکز دارند. الگوهایی که به مکانیسم‌های ایجاد اشیاء مربوط می‌شوند. اینجا همه چیز در مورد instantiation است.

این الگوها عبارتند از:

- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Simple Factory

مثال در دنیای واقعی

در نظر بگیرید، شما مشغول ساخت خانه هستید و به تعدادی درب نیاز دارید. می‌توانید لباس نجاری خود را بپوشید، مقداری چوب، میخ و تمام ابزار مورد نیاز برای ساخت درب را تهیه و شروع به ساختن آن‌ها در خانه خود کنید. یا می‌توانید خیلی ساده با کارخانه/کارگاه تماس و درب‌های ساخته شده را سفارش و تحویل بگیرید تا نیازی به یادگیری چیزی در مورد ساخت درب نداشته باشید.

به بیان ساده

simple factory به سادگی نمونه‌ای را برای شما ایجاد می‌کند بدون اینکه منطق ایجاد و ساخت آن را نشان دهد.

مثال برنامه نویسی

ابتدا یک interface از درب و پیاده‌سازی درب چوبی (implementation) را انجام می‌دهیم.

```
Interface Door
{
   public function getWidth(): float;
   public function getHeight(): float;
}

class WoodenDoor implements Door
{
   protected $width;
   protected $height;

   public function __construct(float $width, float $height)
   {
      $this->width = $width;
      $this->height = $height;
   }

   public function getWidth(): float
   {
      return $this->width;
   }

   public function getHeight(): float
   {
```

```
        return $this->height;
    }
}
```

سپس کارخانه/factory درب که وظیفه آن ساخت درب چوبی است را می‌نویسیم.

```
class DoorFactory
{
    public static function makeDoor($width, $height): Door
    {
        return new WoodenDoor($width, $height);
    }
}
```

و استفاده

```
// Make me a door of 100x200
$door = DoorFactory::makeDoor(100, 200);

echo 'Width: ' . $door->getWidth();
echo 'Height: ' . $door->getHeight();

// Make me a door of 50x100
$door2 = DoorFactory::makeDoor(50, 100);
```

قابلیت استفاده

زمانی که یک شئ با منطق مشخص اما همراه با پارامترهای متفاوت ایجاد می‌شود. بهتر است به جای تکرار این کد در همه جا، آن را در یک factory اختصاصی قرار دهید.

بدون simple factory کلاس WoodenDoor در همه جا تکرار می‌شد.

```
// Make me a door of 100x200
$door = new WoodenDoor(100, 200);
echo 'Width: '. $door->getWidth();
echo 'Height: '. $door->getHeight();

// Make me a door of 50x100
$door2 = new WoodenDoor(50, 100);
```

این عمل تبعات منفی زیادی دارد که در ادامه این کتاب بیشتر آن را درک می‌کنید. به عنوان مثال زمانی که نیاز به تغییر یا جاگزینی کلاس WoodenDoor دارید مجبور خواهید شد هر جایی که از آن استفاده شده است را ویرایش کنید در حالی که با استفاده از الگوی simple factory فقط لازم است کلاس factory اصلاح شود.

Factory Method

مثال در دنیای واقعی

بخش مدیریت استخدام را در نظر بگیرید. وجود یک مصاحبه کننده برای تمام موقعیت‌های شغلی غیر ممکن است. چون مصاحبه کننده به تخصص نیاز دارد.
به همین دلیل بر اساس فرصت‌های شغلی موجود، مدیر استخدام باید در مورد مصاحبه کننده تصمیم بگیرد و این موضوع را به افراد متفاوتی واگذار کند.

به بیان ساده

factory method یک راه برای ایجاد و توسعه اشیاء در subclass ها می‌باشد. زیر کلاس‌ها مسئول ایجاد جزئیات هستند.
در این الگو داخل کلاس interface یا abstract تعریف شده و instantiation در زیر کلاس‌ها انجام می‌شود.

مثال برنامه نویسی

در مثال مدیریت استخدام، ابتدا interface مصاحبه کننده و نمونه‌هایی از آن را برای موقعیت‌های شغلی مورد نظرمان پیاده‌سازی می‌کنیم.

```
interface Interviewer
{
    public function askQuestions();
}

class Developer implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about design patterns!';
    }
}

class CommunityExecutive implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about community building';
    }
}
```

و حالا نوبت پیاده‌سازی مدیر استخدام (HiringManager) است.

```
abstract class HiringManager
{
    // factory method
    abstract protected function makeInterviewer(): Interviewer;

    public function takeInterview()
    {
```

```
        $interviewer = $this->makeInterviewer();
        $interviewer->askQuestions();
    }
}
```

به factory method در کلاس بالا دقت کنید. با این روش کلاس‌های فرزند می‌توانند آن را توسعه دهند.

```
class DevelopmentManager extends HiringManager
{
    protected function makeInterviewer(): Interviewer
    {
        return new Developer();
    }
}

class MarketingManager extends HiringManager
{
    protected function makeInterviewer(): Interviewer
    {
        return new CommunityExecutive();
    }
}
```

و استفاده

```
$devManager = new DevelopmentManager();
$devManager->takeInterview(); // Output: Asking about design patterns

$marketingManager = new MarketingManager();
```

```
$marketingManager->takeInterview(); // Output: Asking about community building.
```

قابلیت استفاده

- زمانیکه وابستگی‌های بین اشیاء نامشخص است و نمی‌دانید دقیقاً چه زیر کلاسی ممکن است نیاز باشد.
- وقتی چندین کلاس با منطق مشترک دارید و به دنبال راهی برای توسعه اجزای داخلی کلاس‌های اصلی هستید.

factory method باعث می‌شود یک طراحی قابل تنظیم‌تر اما پیچیده داشته باشید. این پیچیدگی بد نیست، چون با استفاده از آن به جای نیاز به ایجاد کلاس‌ها فقط عملیات جدید را پیاده‌سازی می‌کنید.

Abstract Factory

مثال در دنیای واقعی

مثال درب چوبی را ادامه می‌دهیم. شما یک درب چوبی از فروشگاه درب‌های چوبی، درب آهنی از فروشگاه درب‌های آهنی و همچنین درب PVC از فروشگاه مرتبط تهیه می‌کنید.
به علاوه اینکه نیاز به متخصص برای نصب درب‌ها دارین. نجار برای درب چوبی و جوشکار برای درب آهنی.
همانطور که می‌بینید بین درب‌ها وابستگی‌هایی وجود دارد (نجار و جوشکار).

تعریف concrete class

یک کلاس که تمام متدهای آن پیاده‌سازی شده. اونها متدهای غیر قابل اجرا ندارند.

به بیان ساده

abstract factory امکان ایجاد خانواده‌ای از اشیاء مرتبط بدون concrete class آن‌ها را فراهم می‌کند و به ما این امکان را می‌دهد که برای ایجاد اشیاء از یک الگوی کلی پیروی کنیم.

مثال برنامه نویسی

در ابتدا یک interface از درب و پیاده‌سازی‌هایی از آن را داریم (چوبی و آهنی).

```
interface Door
{
   public function getDescription();
}

class WoodenDoor implements Door
{
   public function getDescription()
   {
      echo 'I am a wooden door';
   }
}

class IronDoor implements Door
{
   public function getDescription()
   {
      echo 'I am an iron door';
   }
}
```

سپس برای هر نوع درب متخصص مناسب را خواهیم داشت (نجار و جوشکار).

```
interface DoorFittingExpert
{
   public function getDescription();
}

class Welder implements DoorFittingExpert
```

```
{
    public function getDescription()
    {
        echo 'I can only fit iron doors';
    }
}

class Carpenter implements DoorFittingExpert
{
    public function getDescription()
    {
        echo 'I can only fit wooden doors';
    }
}
```

تا اینجا کلاس‌های وابسته/مرتبط را پیاده‌سازی کردیم. اکنون abstract factory را می‌نویسیم که امکان تشکیل خانواده اشیاء مرتبط را برای ما فراهم می‌کند. کارخانه درب چوبی یک درب و متخصص درب چوبی را ایجاد می‌کند و کارخانه درب آهنی هم یک درب و متخصص درب آهنی را می‌سازد.

```
interface DoorFactory
{
    public function makeDoor(): Door;
    public function makeFittingExpert(): DoorFittingExpert;
}
// Wooden factory(concrete factory) to return carpenter and wooden door
class WoodenDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
```

```
        return new WoodenDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Carpenter();
    }
}

// Iron door factory(concrete factory) to get iron door and the relevant
fitting expert
class IronDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
        return new IronDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Welder();
    }
}
```

و استفاده

```
$woodenFactory = new WoodenDoorFactory();
$door = $woodenFactory->makeDoor();
$expert = $woodenFactory->makeFittingExpert();
$door->getDescription();// Output: I am a wooden door
$expert->getDescription(); // Output: I can only fit wooden doors
```

```
// Same for Iron Factory
$ironFactory = new IronDoorFactory();
$door = $ironFactory->makeDoor();
$expert = $ironFactory->makeFittingExpert();
$door->getDescription();// Output: I am an iron door
$expert->getDescription(); // Output: I can only fit iron doors
```

قابلیت استفاده

- در یک طراحی خوب هر کلاس فقط مسئول یک چیز است. زمانیکه یک کلاس با انواع مختلفی از محصولات سروکار دارد نیاز به پیاده‌سازی این الگو احساس خواهد شد.
- هنگامی که سیستم نیاز به مستقل بودن از چگونگی ایجاد، ترکیب و نمایش آن دارد.
- وقتی ایجاد اشیاء مرتبط به سادگی امکانپذیر نیست. در صورتی که وابستگی‌های متقابل بین اشیاء وجود دارد instantiation آن‌ها به صورت تکی/جداگانه منطقی خواهد بود.

Builder

مثال در دنیای واقعی

تصور کنید در فست فود یک غذای ویژه را سفارش می‌دهید. میگویید پیشنهاد سرآشپز! و آن‌ها بدون هیچ سؤالی سفارش شما را انجام و تحویل می‌دهند. این یک مثال از simple factory بود. اما مواردی وجود دارد که برای instantiation مراحل بیشتری نیاز داریم.

مثلاً برای سفارش همبرگر چه نوع نانی را می‌خواهید؟ چه سس و یا پنیری و... در چنین مواردی builder به کمک ما می‌آید.

به بیان ساده

به شما این اماکن را می‌دهد تا ضمن جلوگیری از کثیف کردن constructor، انواع مختلفی از یک شئ را ایجاد کنید.

دقیقاً منظورمان چیست؟ همه ما تابع سازنده‌ای مثل این نمونه را دیده‌ایم (یک نمونه کثیف).

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

همانطور که می‌بینید پارامترهای تابع زیاد و ممکن است تعداد آن از دست برنامه نویس خارج شود و یا اینکه ترتیب چیدمان آن را فراموش کند. همچنین ممکن است به دلیل توسعه، لیست پارمترها افزایش یابد. اصطلاحاً به این موضوع telescoping constructor anti-pattern میگویند.

مثال برنامه نویسی

جایگزین مناسب استفاده از الگوی builder است. در ابتدا می‌خواهیم همبرگر خود را درست کنیم.

```
class Burger
{
   protected $size;
   protected $cheese = false;
   protected $pepperoni = false;
   protected $lettuce = false;
   protected $tomato = false;

   public function __construct(BurgerBuilder $builder)
   {
      $this->size = $builder->size;
      $this->cheese = $builder->cheese;
      $this->pepperoni = $builder->pepperoni;
      $this->lettuce = $builder->lettuce;
      $this->tomato = $builder->tomato;
   }
}
```

و سپس builder همبرگر را می‌نویسیم.

```
class BurgerBuilder
{
    public $size;
    public $cheese = false;
    public $pepperoni = false;
    public $lettuce = false;
    public $tomato = false;

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

    public function addPepperoni()
    {
        $this->pepperoni = true;
        return $this;
    }

    public function addLettuce()
    {
        $this->lettuce = true;
        return $this;
    }

    public function addCheese()
    {
        $this->cheese = true;
        return $this;
    }
```

```
    public function addTomato()
    {
        $this->tomato = true;
        return $this;
    }

    public function build(): Burger
    {
        return new Burger($this);
    }
}
```

و استفاده

```
$burger = (new BurgerBuilder(14))
->addPepperoni()
->addLettuce()
->addTomato()
->build();
```

قابلیت استفاده

- برای حل مشکل telescopic constructor.
- هنگامی که از یک شئ انواع مختلفی ساخته می‌شود. تفاوت آن با الگوی factory ایجاد اشیاء در چندین گام/مرحله است.

Prototype

مثال در دنیای واقعی

dolly را به یاد دارید؟ گوسفندی که همانندسازی شده بود. اجازه بدهید وارد جزئیات نشوم. اینجا همه چیز در مورد cloning است.

https://en.wikipedia.org/wiki/Dolly_(sheep)

به بیان ساده

ایجاد شئ از یک شئ موجود با استفاده از cloning.

به جای اینکه یک شئ جدید ایجاد کنید و از ابتدا تنظیمات آن را انجام دهید، اجازه می‌دهد یک کپی از شئ موجود را ایجاد کرده و آن را با توجه به نیاز خود تغییر دهید.

مثال برنامه نویسی

کلاس Sheep را در نظر بگیرید.

```
class Sheep
{
   protected $name;
   protected $category;
   public function __construct(string $name, string $category =
'Mountain Sheep')
   {
      $this->name = $name;
      $this->category = $category;
   }

   public function setName(string $name)
   {
      $this->name = $name;
   }

   public function getName()
   {
      return $this->name;
   }
```

```
    public function setCategory(string $category)
    {
        $this->category = $category;
    }

    public function getCategory()
    {
        return $this->category;
    }
}
```

در php می‌توان به راحتی با دستور clone این کار را انجام داد.

```
$original = new Sheep('Jolly');
echo $original->getName(); // Jolly
echo $original->getCategory(); // Mountain Sheep

// Clone and modify what is required
$cloned = clone $original;
$cloned->setName('Dolly');
echo $cloned->getName(); // Dolly
echo $cloned->getCategory(); // Mountain sheep
```

قابلیت استفاده

زمانیکه نیاز به یک شئ جدید مانند شئ موجود داریم، استفاده نکردن از cloning زمانبر خواهد بود.

Singleton

مثال در دنیای واقعی

در یک زمان تنها یک رئیس جمهور برای کشور وجود دارد. و او باید به وظایفش عمل کند. رئیس جمهور در اینجا singleton است.

به بیان ساده

این الگو ایجاد شئ از یک کلاس را محدود می‌کند و اطمینان می‌دهد که فقط یک نمونه از آن کلاس وجود دارد. این الگو زمانی مفید است که در سرتاسر سیستم تنها به یک نمونه از آن کلاس نیاز باشد.

نکته

singleton در واقع به عنوان یک anti-pattern مطرح شده و باید از استفاده بیش از حد آن جلوگیری کرد! لزوماً بد نیست، و می‌تواند برای مواردی معتبر باشد. اما باید مراقب باشید چون تغییر آن در یک محل می‌تواند قسمت‌های دیگر را هم تحت تأثیر قرار دهد و رفع اشکال آن بسیار مشکل شود. نکته بد دیگر این است که باعث tight coupling در کد شما می‌شود.

خیلی کوتاه، tight coupling زمانی اتفاق خواهد افتاد که گروهی از کلاس‌ها بسیار وابسته به یکدیگر باشند.

مثال برنامه نویسی

برای ایجاد singleton یک متد constructor از نوع private بسازید. cloning و extension را غیر فعال و به عنوان مثال یک متغییر static تعریف کنید.

```
final class President
{
   private static $instance;

   private function __construct()
   {
```

```
        // Hide the constructor
    }

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

    private function __clone()
    {
        // Disable cloning
    }

    private function __wakeup()
    {
        // Disable unserialize
    }
}
```

و استفاده

```
$president1 = President::getInstance();
$president2 = President::getInstance();

var_dump($president1 === $president2); // true
```

قابلیت استفاده

برای کلاس‌هایی که می‌بایست فقط یک بار ایجاد شوند و در دسترس همه قرار بگیرند. مثلاً یک شئ از پایگاه داده که در جاهای مختلف نرم افزار به اشتراک گذاشته می‌شود. همچنین برای کنترل دقیقتر بر متغییرهای عمومی نیز کاربرد دارد.

فصل سوم

الگوهای طراحی ساختاری / Structural

الگوهای طراحی ساختاری / Structural

به بیان ساده

structural design pattern ها به ترکیب اشیاء و کلاس‌ها در ساختارهای بزرگ‌تر همراه با حفظ کارامدی و انعطاف پذیری آن ساختارها می‌پردازند. این الگوها به سؤال چگونه اجزای یک نرم افرار را باید ساخت پاسخ می‌دهند. الگوهای طراحی ساختاری با شناسایی روابط بین موجودیت‌ها، ساختار را ساده‌تر می‌کنند و بر چگونگی ارث بری و نحوه تشکیل آن‌ها از یکدیگر تمرکز دارند.

7 نوع از الگوهای structural وجود دارد.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter

مثال در دنیای واقعی.

تبدیل شارژر تلفن همراه را در نظر بگیرید. یک شارژر با پایه سه شاخه که در جعبه تلفن همراه موجود است را نمی‌توانید به پریز دو شاخه منزل وصل کنید. نیاز به پایه تبدیل دارید تا این دو را با هم سازگار کند. این وسیله تبدیل یک adapter است.

به بیان ساده

به اشیاء با رابط‌های ناسازگار امکان همکاری می‌دهد.

این الگو، یک شئ غیر سازگار را در یک adapter برای سازگاری با سایر کلاس‌ها پنهان می‌کند. adapter این اجازه را می‌دهد که دو یا چند شئ ناسازگار با هم ارتباط یا تعامل برقرار کنند.
معمولاً با این هدف مورد استفاده قرار می‌گیرد که بدون تغییر در کد اصلی، بتوان استفاده از کلاس‌های فعلی را مقدور ساخت.

مثال برنامه نویسی

بازی شکارچی شیر را در نظر بگیرید. ابتدا یک interface از شیر داریم که با استفاده از آن انواع شیرها (با نژادهای مختلف) را پیاده‌سازی می‌کنیم.

```
interface Lion
{
   public function roar();
}

class AfricanLion implements Lion
{
   public function roar()
   {
   }
}

class AsianLion implements Lion
{
   public function roar()
   {
   }
}
```

و شکارچی که البته برایش فرقی نمی‌کند کدام نژاد از شیر را شکار خواهد کرد. او برای شکار به دنبال صدای غرش شیر است.

```
class Hunter
{
   public function hunt(Lion $lion)
   {
      $lion->roar();
   }
}
```

حالا بیایید سگ‌های وحشی را هم اضافه کنیم تا شکارچی بتواند آن‌ها را هم شکار کند. اما یک مشکل داریم. سگ مانند شیر غرش نمی‌کند، پارس می‌کند و در فضای برنامه نویسی یعنی سگ interface متفاوتی دارد. اینجا برای سازگاری سگ و شیر از adapter استفاده می‌کنیم.

```
// This needs to be added to the game
class WildDog
{
   public function bark()
   {
   }
}

// Adapter around wild dog to make it compatible with our game
class WildDogAdapter implements Lion
{
   protected $dog;

   public function __construct(WildDog $dog)
```

```
    {
        $this->dog = $dog;
    }

    public function roar()
    {
        $this->dog->bark();
    }
}
```

بدین صورت ناسازگاری کلاس سگ‌های وحشی را (متد پارس سگ) در Adapter پنهان کردیم تا سایر مؤلفه‌های مرتبط بدون مشکل کار خود را انجام دهند.

```
$wildDog = new WildDog();

$wildDogAdapter = new WildDogAdapter($wildDog);
$hunter = new Hunter();
$hunter->hunt($wildDogAdapter);
```

قابلیت استفاده

برای استفاده از کلاس‌هایی که دارای روابط ناسازگار هستند یا functionality مشترکی با هم ندارند.

Bridge

مثال در دنیای واقعی

در نظر بگیرید یک وبسایت با صفحات مختلف دارید و قرار است امکان تغییر قالب وبسایت را در اختیار کاربران قرار دهید. چه کاری انجام می‌دهید؟ به تعداد قالب‌ها از

صفحات کپی و روی هر مجموعه از صفحات یک قالب را اعمال می‌کنید یا هنگام بارگزاری صفحات محتوای آن‌ها را در قالب مورد نظر کاربر قرار و نمایش می‌دهید؟

به بیان ساده

رابط شئ را از پیاده‌سازی آن جدا می‌کند. به طوری که این دو بتوانند به طور مستقل تغییر پذیر باشند.

این الگو به ترکیب اشیاء جای وراثت می‌پردازد. پیاده‌سازی شئ مستقل انجام شده و به اشیاء دیگر منتقل می‌شود.

مثال برنامه نویسی

بر اساس مثال این بخش ابتدا صفحات را پیاده‌سازی می‌کنیم.

```
interface WebPage
{
   public function __construct(Theme $theme);
   public function getContent();
}

class About implements WebPage
{
   protected $theme;

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

   public function getContent()
   {
      return "About page in " . $this->theme->getColor();
   }
```

```
}

class Careers implements WebPage
{
    protected $theme;

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

    public function getContent()
    {
        return "Careers page in " . $this->theme->getColor();
    }
}
```

و سپس قالب‌ها

```
interface Theme
{
    public function getColor();
}

class DarkTheme implements Theme
{
    public function getColor()
    {
        return 'Dark Black';
    }
}
```

```
class LightTheme implements Theme
{
   public function getColor()
   {
      return 'Off white';
   }
}

class AquaTheme implements Theme
{
   public function getColor()
   {
      return 'Light blue';
   }
}
```

همانطور که از نحوه استفاده پیداست اشیاء مستقل از هم ایجاد شده‌اند.

```
$darkTheme = new DarkTheme();

$about = new About($darkTheme);
$careers = new Careers($darkTheme);

echo $about->getContent(); // "About page in Dark Black";
echo $careers->getContent(); // "Careers page in Dark Black";
```

قابلیت استفاده

- هنگامی که هم رابط شئ و هم پیاده‌سازی آن نیاز به توسعه توسط کلاس‌های فرزند دارند.

- برای سازماندهی و یکپارچه سازی کلاس‌ها.bridge این امکان را می‌دهد تا کلاس‌های بزرگ را در یک ساختار سلسله مراتبی پیاده‌سازی کنید. در این ساختار امکان تغییر و توسعه کلاس‌ها به صورت مستقل وجود خواهد داشت و نگهداری از کد را ساده‌تر می‌کند.

Composite

مثال در دنیای واقعی

یک سازمان از کارمندان تشکیل شده. هر یک از کارمندان ویژگی‌های مشابه ای دارند (البته با مقادیر متفاوت). مثلاً ویژگی‌های حقوق و دستمزد، مسئولیت‌ها، اینکه نیاز است به شخص دیگری گزارش دهد یا نه، زیردستان و...

به بیان ساده

این امکان را می‌دهد که با اشیاء به یک شکل رفتار شود. در واقع composite گروهی از اشیاء را در یک ساختار درختی یا سلسله مراتبی توصیف می‌کند که این اشیا به خاطر یکسان بودن نوعشان به یک شکل با آن‌ها رفتار می‌شود در حالی که انگار اشیاء مستقل و جداگانه‌ای هستند.

مثال برنامه نویسی

بر اساس مثال بالا ما در اینجا انواع کارمند داریم (برنامه نویس، طراح و...).

```
interface Employee
{
   public function __construct(string $name, float $salary);

   public function getName(): string;
   public function setSalary(float $salary);
   public function getSalary(): float;
   public function getRoles(): array;
```

```
}

class Developer implements Employee
{
    protected $salary;
    protected $name;
    protected $roles;

    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
```

```
    }
}

class Designer implements Employee
{
    protected $salary;
    protected $name;
    protected $roles;
    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }

    public function getName(): string
    {
        return $this->name;
    }
    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
    }
```

```
}
```

سپس سازمانی که از چندین کارمند با تخصص‌های مختلف تشکیل شده.

```
class Organization
{
    protected $employees;
    public function addEmployee(Employee $employee)
    {
        $this->employees[] = $employee;
    }
    public function getNetSalaries(): float
    {
        $netSalary = 0;
        foreach ($this->employees as $employee) {
            $netSalary += $employee->getSalary();
        }
        return $netSalary;
    }
}
```

نحوه استفاده

```
// Prepare the employees
$john = new Developer('John Doe', 12000);
$jane = new Designer('Jane Doe', 15000);

// Add them to organization
$organization = new Organization();
$organization->addEmployee($john);
```

```
$organization->addEmployee($jane);

echo "Net salaries: ". $organization->getNetSalaries(); // Net Salaries: 27000
```

قابلیت استفاده

زمانی که امکان پیاده‌سازی یک ساختار درختی/سلسله مراتبی از اشیاء وجود دارد، composite اضافه کردن مؤلفه‌های (components) جدید را برای شما آسان می‌کند بدون آنکه روی سایر اشیاء تأثیر بگذارد.

Decorator

مثال در دنیای واقعی

یک فروشگاه خدمات خودرو با ارائه چندین سرویس را در نظر بگیرید. چطور صورت حساب مشتریان را محاسبه می‌کنید؟

شما یک سرویس را انتخاب می‌کنید و هزینه آن به طور خودکار به مبلغ خدمات ارائه شده مشتری اضافه می‌شود تا زمانی که هزینهٔ نهایی مشخص گردد. در اینجا هر سرویس یک decorator است.

به بیان ساده

به شما اجازه می‌دهد رفتار یک شئ را به طور پویا در زمان اجرا توسط کلاس decorator **تغییر** دهید. این الگو از ترکیب به جای وراثت استفاده می‌کند.

decorator امکان افزودن رفتار (behavior) به یک شئ، را به‌طور پویا (dynamic) فراهم می‌سازد بی آنکه رفتار اشیاء دیگر از همان کلاس (که شئ مورد بحث از آن ساخته شده) دست‌خوش تغییر شوند.

مثال برنامه نویسی

اجازه بدهید یک مثال در مورد نوشیدن قهوه بزنیم. در ابتدا ما یک قهوه ساده داریم. پس interface و پیاده‌سازی‌های آن را انجام می‌دهیم.

```
interface Coffee
{
   public function getCost();
   public function getDescription();
}

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

   public function getDescription()
   {
      return 'Simple coffee';
   }
}
```

خب برای جذب مشتریان انواع مختلفی از قهوه را آماده می‌کنیم. قهوه با شیر، وانیل و...

```
class MilkCoffee implements Coffee
{
   protected $coffee;

   public function __construct(Coffee $coffee)
```

```
    {
        $this->coffee = $coffee;
    }

    public function getCost()
    {
        return $this->coffee->getCost() + 2;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', milk';
    }
}

class WhipCoffee implements Coffee
{
    protected $coffee;

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

    public function getCost()
    {
        return $this->coffee->getCost() + 5;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', whip';
```

```
    }
}

class VanillaCoffee implements Coffee
{
    protected $coffee;

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

    public function getCost()
    {
        return $this->coffee->getCost() + 3;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', vanilla';
    }
}
```

حالا بیایید قهوه درست کنیم.

```
$someCoffee = new SimpleCoffee();
echo $someCoffee->getCost(); // 10
echo $someCoffee->getDescription(); // Simple Coffee

$someCoffee = new MilkCoffee($someCoffee);
echo $someCoffee->getCost(); // 12
```

```
echo $someCoffee->getDescription(); // Simple Coffee, milk

$someCoffee = new WhipCoffee($someCoffee);
echo $someCoffee->getCost(); // 17
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip

$someCoffee = new VanillaCoffee($someCoffee);
echo $someCoffee->getCost(); // 20
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip, vanilla
```

قابلیت استفاده

هنگامی که نیاز است به طور پویا اشیاء در زمان اجرا تغییر کنند بدون اینکه روی دیگر اشیاء تأثیر داشته باشند.

decorator از ترکیب به جای وراثت استفاده می‌کند چون وراثت ایستا است. شما نمی‌توانید رفتار یک شئ موجود را در زمان اجرا تغییر دهید. همچنین در اکثر زبان‌ها وراثت اجازه نمی‌دهد یک کلاس رفتار چند کلاس را به طور هم‌زمان به ارث ببرد.

Facade

مثال در دنیای واقعی

چطور کامپیوتر را روشن می‌کنید؟ دکمه power را می‌زنید. شما از یک رابط کاربری ساده استفاده می‌کنید که کامپیوتر آن را خارج از خود ارائه داده اما برای روشن شدن در داخل نیز باید کارهای زیادی انجام دهد. دکمه power یک رابط کاربری ساده برای سیستم پیچیده‌ای است که به آن facade میگوییم.

به بیان ساده

یک رابط کاربری ساده برای سیستم پیچیده فراهم می‌کند.

معمولاً هنگامی از الگوی طراحی facade استفاده می‌شود که سیستم از پیچیدگی‌های زیادی برخوردار یا درک آن دشوار باشد، این الگو پیچیدگی‌ها را مخفی کرده و یک رابط ساده برای توسعه دهنده فراهم می‌کند.

مثال برنامه نویسی

همان مثال بالا در مورد کامپیوتر را انجام می‌دهیم.

```
class Computer
{
   public function getElectricShock()
   {
      echo "Ouch!";
   }

   public function makeSound()
   {
      echo "Beep beep!";
   }

   public function showLoadingScreen()
   {
      echo "Loading..";
   }

   public function bam()
   {
      echo "Ready to be used!";
   }

   public function closeEverything()
   {
```

```
        echo "Bup bup bup buzzzz!";
    }

    public function sooth()
    {
        echo "Zzzzz";
    }

    public function pullCurrent()
    {
        echo "Haaah!";
    }
}
```

و برای رهایی از این پیچیدگی الگوی facade را پیاده‌سازی می‌کنیم.

```
class ComputerFacade
{
    protected $computer;

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

    public function turnOn()
    {
        $this->computer->getElectricShock();
        $this->computer->makeSound();
        $this->computer->showLoadingScreen();
        $this->computer->bam();
```

```
    }

    public function turnOff()
    {
        $this->computer->closeEverything();
        $this->computer->pullCurrent();
        $this->computer->sooth();
    }
}
```

و استفاده

```
$computer = new ComputerFacade(new Computer());
$computer->turnOn(); // Ouch! Beep beep! Loading.. Ready to be used!
$computer->turnOff(); // Bup bup buzzz! Haah! Zzzzz
```

قابلیت استفاده

زمانی که می‌خواهید رابط ساده برای یک سیستم پیچیده ایجاد کنید.

Flyweight

مثال در دنیای واقعی

تا به حال در قهوه خانه چای سفارش داده‌اید؟ قهوه‌چی معمولاً بین میزها قدم می‌زند و بیشتر از یک سفارش تحویل می‌گیرد. سپس با یک سینی از انواع چای برمی‌گردد (سفارشات). او با این کار در زمان و انرژی صرفه جویی می‌کند.

قاعدتاً آماده و سِرو کردن یک سینی چای برای چند نفر نسبت به گرفتن و تحویل دادن تکی سفارش انرژی و زمان کمتری لازم دارد. flyweight به همین مورد می‌پردازد. اشتراک گذاری.

به بیان ساده

از اشتراک گذاری قسمت‌های مشابه برای به حداقل رساندن استفاده از منابع مانند ram بهره می‌برد.
این الگو حالت‌های تکراری بین اشیاء را از بین می‌برد، در منابع صرفه جویی می‌کند و تعداد اشیاء را نیز کاهش می‌دهد.
یکی از ویژگی‌های اشیاء flyweight این است که آن‌ها تغییر ناپذیرند و پس از ساخته شدن نمی‌توانند اصلاح شوند.

در ویکی پدیا گفته شده

یک شئ flyweight نوعی شئ است که باعث صرفه جویی در مصرف حافظه می‌شود. یک شئ flyweight تا جایی که ممکن هست با اشیا همنوع خود به اشتراک اطلاعات می‌پردازد. وقتی که تعداد زیادی از یک نوع شئ دارای اشتراک اطلاعاتی باشند، تکرار این قسمت مشترک در هر کدام از این نمونه اشیاء باعث مصرف زیاد از حد حافظه خواهد شد که غیرقابل قبول است. برای حل این مشکل معمولاً قسمت مشترک را در یک داده ساختار دیگر قرار داده که این اشیاء سبک وزن همگی به آن دسترسی خواهند داشت.

مثال برنامه نویس

مثال همین قسمت را انجام می‌دهیم. ابتدا چای و چای ساز را پیاده‌سازی می‌کنیم.

```
// Anything that will be cached is flyweight.
// Types of tea here will be flyweights.
class KarakTea
{
}

// Acts as a factory and saves the tea
class TeaMaker
{
   protected $availableTea = [];
```

```
    public function make($preference)
    {
        if (empty($this->availableTea[$preference])) {
            $this->availableTea[$preference] = new KarakTea();
        }
        return $this->availableTea[$preference];
    }
}
```

سپس فروشگاه نوشیدنی/چای که سفارش می‌گیرد و سرویس می‌دهد.

```
class TeaShop
{
    protected $orders;
    protected $teaMaker;

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

    public function takeOrder(string $teaType, int $table)
    {
        $this->orders[$table] = $this->teaMaker->make($teaType);
    }

    public function serve()
    {
        foreach ($this->orders as $table => $tea) {
            echo "Serving tea to table# " . $table;
```

```
        }
    }
}
```

و استفاده

```
$teaMaker = new TeaMaker();
$shop = new TeaShop($teaMaker);

$shop->takeOrder('less sugar', 1);
$shop->takeOrder('more milk', 2);
$shop->takeOrder('without sugar', 5);

$shop->serve();
// Serving tea to table# 1
// Serving tea to table# 2
// Serving tea to table# 5
```

قابلیت استفاده

برای صرفه جویی در منابع در زمانی که اشیاء مشابه زیادی داریم.

Proxy

مثال در دنیای واقعی

آیا تا به حال از کارت برای عبور از درب‌های امنیتی استفاده کرده‌اید؟ اگر نه حتماً در فیلم‌های سینمایی دیده‌اید. روش‌های مختلفی برای باز کردن درب وجود دارد. می‌توانید با کارت یا وارد کردن کلمه عبور با استفاده از صفحه کلیدی که روی درب به عنوان گزینهٔ فرعی نصب شده این کار را انجام دهید.

به بیان ساده

یک شئ که نمایانگر شئ دیگر است. الگوی proxy دسترسی به شئ اصلی را فراهم می‌کند و به شما این امکان را می‌دهد که قبل یا بعد از درخواست به شئ اصلی عملیات اضافی دیگری را انجام دهید. مثلاً کنترل دسترسی کاربران یا مخفی کردن اطلاعات شئ اصلی و....

در ویکی پدیا گفته شده

در کلی‌ترین حالت، کلاس واسطه‌ای برای انجام کار دیگری است. proxy می‌تواند رابط هر چیزی باشد که نمونه‌سازی از آن بسیار سخت یا غیرممکن است. به صورت خلاصه، proxy وسیله‌ای است که توسط مشتری صدا زده می‌شود تا به شئ اصلی در پشت‌پرده دسترسی پیدا کند. استفاده از proxy می‌تواند انتقال به شئ واقعی را راحت‌تر کند یا روش‌های دیگری برای آن به وجود بیاورد. در الگوی proxy قابلیت‌های اضافه‌تر می‌تواند به وجود بیاید، به‌طور مثال، استفاده از حافظه موقت، برای وقتی که استفاده از شئ اصلی هزینه پردازشی و هزینه دسترسی به حافظه زیادی داشته باشد، یا در چک کردن شرایط اولیه قبل از آن که عملیات بر روی شئ اصلی اعمال شود. برای مشتری با استفاده از وکیل مانند استفاده از شئ اصلی است، چرا که رابط هر دو مشابه است.

مثال برنامه نویسی

مثال درب امنیتی را پیاده می‌کنیم. ابتدا interface درب و یک پیاده‌سازی از آن را داریم.

```
interface Door
{
   public function open();
   public function close();
}

class LabDoor implements Door
{
```

```
    public function open()
    {
        echo "Opening lab door";
    }

    public function close()
    {
        echo "Closing the lab door";
    }
}
```

و سپس proxy درب امنیتی

```
class SecuredDoor
{
    protected $door;

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

    public function open($password)
    {
        if ($this->authenticate($password)) {
            $this->door->open();
        } else {
            echo "Big no! It ain't possible.";
        }
    }
```

```
    public function authenticate($password)
    {
        return $password === '$ecr@t';
    }

    public function close()
    {
        $this->door->close();
    }
}
```

و استفاده

```
$door = new SecuredDoor(new LabDoor());
$door->open('invalid'); // Big no! It ain't possible.
$door->open('$ecr@t'); // Opening lab door
$door->close(); // Closing lab door
```

فصل چهارم

الگوهای طراحی رفتاری / Behavioral

الگوهای طراحی رفتاری / Behavioral

به بیان ساده

الگوهای طراحی رفتاری مربوط به تخصیص مسئولیت‌ها بین اشیاء است. آن‌ها الگوهای ارتباطی مشترک بین اشیاء را شناسایی و پیاده‌سازی می‌کنند. با استفاده از این الگوها انعطاف پذیری در ارتباطات بین اشیاء افزایش پیدا خواهد کرد.

چیزی که باعث تفاوت آن‌ها با الگوهای structural می‌شود این است که آن‌ها علاوه بر مشخص کردن ساختار، انتقال پیام و ارتباطات بین اشیاء را نیز تشریح می‌کنند. به عبارت دیگر، آن‌ها به سؤال چگونه یک رفتار را در اجزای نرم‌افزار اجرا کنیم پاسخ می‌دهند.

10 نوع از این الگوها وجود دارد.

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- Visitor
- Strategy
- State
- Template Method

Chain of Responsibility

مثال در دنیای واقعی

فرض کنید شما سه حساب اعتباری برای پرداخت خریدهای خود دارید که در هر کدام موجودی متفاوتی وجود دارد. حساب A موجودی 100 دلار. B موجودی 300

دلار و C هم 1000 دلار. ترجیح می‌دهید اولویت برداشت از حساب‌هایتان به صورت ابتدا از A و سپس B و در نهایت C باشد.
شما یک خرید 210 دلاری دارید. با استفاده از chain of responsibility، ابتدا حساب A بررسی می‌شود و اگر امکان برداشت وجود داشت (مبلغ خرید کمتر یا برابر با موجودی حساب بود) تمام! در غیر این صورت درخواست به سمت حساب B شما ارسال می‌شود. تا زمانی که برداشت قابل انجام باشد این فرآیند ادامه خواهد داشت. سه حساب شما یک chain of responsibility هستند و درواقع هر کدام، یک حلقه زنجیر از آن.

به بیان ساده

راهی برای ارسال درخواست بین زنجیره‌ای از اشیاء می‌باشد. این الگو زنجیره‌ای از اشیاء را ایجاد می‌کند، درخواست وارد و از یک شئ به شئ دیگر منتقل می‌شود. تا زمانی که گیرندهٔ مناسب را پیدا کند.

در ویکی پدیا گفته شده

یک الگوی طراحی متشکل از شئ **فرمانده** و یک سری از اشیای پردازشی است. هر شئ پردازشی می‌تواند وظایف دریافتی را انجام دهد یا به اصطلاح پردازش کند. chain of responsibility الگوی ساختاری تقریباً یکسانی با decorator دارد، با این تفاوت که در decorator همه کلاس‌ها می‌توانند یک درخواست را رسیدگی کنند اما در الگوی chain of responsibility فقط یکی از کلاس‌ها در زنجیره به درخواست رسیدگی می‌کند.

مثال برنامه نویسی

مثال همین بخش را انجام می‌دهیم. کلاس Account، وظایف پرداخت، امکان پرداخت و ایجاد زنجیره را بررسی و انجام می‌دهد. این همان شئ **فرمانده** است. سه حساب اعتباری نیز تعریف می‌کنیم.

```
abstract class Account
{
    protected $successor;
    protected $balance;

    public function setNext(Account $account)
    {
        $this->successor = $account;
    }

    public function pay(float $amountToPay)
    {
        if ($this->canPay($amountToPay)) {
            echo sprintf('Paid %s using %s' . PHP_EOL, $amountToPay,
get_called_class());
        } elseif ($this->successor) {
            echo sprintf('Cannot pay using %s. Proceeding ..' . PHP_EOL,
get_called_class());
            $this->successor->pay($amountToPay);
        } else {
            throw new Exception('None of the accounts have enough
balance');
        }
    }

    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;
    }
}
```

در مرحله بعدی به منظور استفاده، زنجیره اولویت را تعریف می‌کنیم و سپس عملیات پرداخت را انجام می‌دهیم.

```
// Let's prepare a chain like below
// $bank->$paypal->$bitcoin
//
```

```
// First priority bank
// If bank can't pay then paypal
// If paypal can't pay then bit coin
$bank = new Bank(100);// Bank with balance 100
$paypal = new Paypal(200);// Paypal with balance 200
$bitcoin = new Bitcoin(300);// Bitcoin with balance 300

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

// Let's try to pay using the first priority i.e. bank
$bank->pay(259);
// Output will be
// ==============
// Cannot pay using bank. Proceeding..
// Cannot pay using paypal. Proceeding..:
// Paid 259 using Bitcoin!
```

قابلیت استفاده

وقتی انتظار دارید برنامه شما درخواست‌ها را به روش‌های مختلف پاسخ دهد. یا زمانی که لازم است برای درخواست‌ها ترتیب خاصی رعایت شود.

Command

مثال در دنیای واقعی

شما در رستوران مشغول سفارش غذا هستید. شما (Client/کاربر) از پیش خدمت (Invoker/معرف) می‌خواهید تا غذا بیاورد (Command/دستور). پیشخدمت درخواست را به آشپز می‌دهد (Receiver/گیرنده) و او می‌داند برای سفارش شما چه چیزهایی را آماده کند.

به بیان ساده

درخواست را به یک شئ مستقل تحت عنوان command تبدیل می‌کند. هدف این الگو ارائه ابزاری برای جدا کردن کاربر از گیرنده است.

در ویکی پدیا گفته شده

در این الگو از یک شئ به منظور مخفی کردن اطلاعات مورد نیاز برای انجام یک عمل یا اجرای رویدادی در زمان آینده، استفاده می‌شود. این اطلاعات شامل؛ نام روش، شئ مالک روش و مقادیر پارامترهای مربوط به روش می‌باشد. بخش‌های مرتبط با الگوی فرمان اغلب از چهار بخش؛ دستور فرمان، گیرنده فرمان، معرف فرمان و کاربر تشکیل می‌شود.

مثال برنامه نویسی

ابتدا ما یک گیرنده و اعمالی که می‌خواهیم انجام دهیم را پیاده‌سازی می‌کنیم. مثلاً گیرنده‌ای به نام لامپ که دو عمل خاموش و روشن شدن را دارد.

```
// Receiver
class Bulb
{
   public function turnOn()
   {
      echo "Bulb has been lit";
   }

   public function turnOff()
   {
      echo "Darkness!";
   }
}
```

سپس یک interface از دستورات اجرایی را می‌نویسیم. دقت کنید، در اینجا command ها شئ هستند.

```
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();
    }

    public function redo()
    {
        $this->execute();
    }
}
```

مرحله بعدی پیاده‌سازی Invoker است.Invoker با کاربر برای پردازش دستورات تعامل دارد.

```
// Invoker
class RemoteControl
{
    public function submit(Command $command)
```

```
    {
        $command->execute();
    }
}
```

و استفاده

```
$bulb = new Bulb();

$turnOn = new TurnOn($bulb);
$turnOff = new TurnOff($bulb);

$remote = new RemoteControl();
$remote->submit($turnOn); // Bulb has been lit!
$remote->submit($turnOff); // Darkness!
```

قابلیت استفاده

با استفاده از الگوی command می‌توانید درخواست را به یک شئ تبدیل کنید. این تبدیل کاربردهای زیادی دارد:

- قابلیت ارسال command به صورت آرگومان در متدها.
- ذخیره سازی در اشیاء دیگر.
- سوئیچ بین command ها در زمان اجر.
- استفاده برای عملیات logging،rollback و...

Iterator

مثال در دنیای واقعی

رادیوهای قدیمی مثال خوبی برای Iterator هستند. می‌توانستیم از یک محل شروع کنیم (که معمولاً قبلاً خودمان یا شخص دیگری آن را تنظیم کرده بود) و با استفاده از پیچاندن غلطک رادیو در میان فرکانس‌ها به ایستگاه‌های رادیویی سوئیچ کنیم.

به بیان ساده

راهی برای دسترسی به عناصر مجموعه‌ای از اشیاء بدون دانستن جزئیات پیاده‌سازی و پیچیدگی‌های آنها.

در ویکی پدیا گفته شده

در این الگو Iterator با هدف عبور از یک مجموعه داده و دسترسی به عناصر آن مجموعه، مورد استفاده قرار می‌گیرد. ماهیت الگوهای Iterator اغلب فراهم نمودن راهی برای دسترسی پیوسته به عناصر یک مجموعه داده و جدا نمودن الگوریتم‌ها، بدون دانستن از ساختار یا فاش کردن محتویات آن مجموعه می‌باشد.

مثال برنامه نویسی

مثال این بخش را پیاده‌سازی می‌کنیم. در ابتدا یک ایستگاه رادیویی داریم.

```
class RadioStation
{
   protected $frequency;

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

   public function getFrequency(): float
   {
      return $this->frequency;
   }
}
```

سپس iterator را پیاده‌سازی می‌کنیم. لیست ایستگاه‌های رادیویی. در PHP با استفاده از SPL یا Standard PHP Library آماده کردن iterator بسیار آسان است.

```
use Countable;
use Iterator;

class StationList implements Countable, Iterator
{
    /** @var RadioStation[] $stations */
    protected $stations = [];
    /** @var int $counter */
    protected $counter;

    public function addStation(RadioStation $station)
    {
        $this->stations[] = $station;
    }

    public function removeStation(RadioStation $toRemove)
    {
        $toRemoveFrequency = $toRemove->getFrequency();
        $this->stations = array_filter($this->stations, function (RadioStation
$station) use ($toRemoveFrequency) {
            return $station->getFrequency() !== $toRemoveFrequency;
        });
    }

    public function count(): int
    {
        return count($this->stations);
    }

    public function current(): RadioStation
    {
```

```
        return $this->stations[$this->counter];
    }

    public function key()
    {
        return $this->counter;
    }

    public function next()
    {
        $this->counter++;
    }

    public function rewind()
    {
        $this->counter = 0;
    }

    public function valid(): bool
    {
        return isset($this->stations[$this->counter]);
    }

}
```

و در نهایت بدین صورت ایستگاه‌های رادیویی را اضافه و استفاده می‌کنیم.

```
$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)); // Will remove station 89
```

قابلیت استفاده

هنگامی که مجموع ای از اشیاء با داده و ساختارهای پیچیده دارید و می‌خواهید به هر دلیلی (مثلاً امنیت یا سادگی و..) پیچیدگی‌ها را از دید توسعه دهنده پنهان کنید.
یا وقتی که می‌خواهید قابلیت عبور از ساختار داده‌ای متفاوت را داشته باشید. مخصوصاً زمانی که این ساختارها از قبل ناشناخته هستند.

Mediator

مثال در دنیای واقعی

در هنگام صحبت تلفنی، شبکه مخابرات بین شما و طرف دیگر وجود دارد که مکالمات شما از طریق این شبکه منتقل می‌شود. در این مثال شبکه مخابرات یک mediator است.

به بیان ساده

روابط مجموعه‌ای از اشیاء را تهیه و تعیین می‌کند و وابستگی بین اشیاء را کاهش می‌دهد.
mediator ارتباطات مستقیم بین اشیاء را محدود کرده و آن‌ها را وادار به تعامل تنها از طریق شئ mediator می‌کند.
این الگو یک شئ ثالث یا third party که mediator نام دارد را به منظور کنترل تعامل بین دو یا چند شئ که به آن‌ها همکاران یا colleagues گفته می‌شود اضافه می‌کند. این موضوع باعث کاهش coupling بین کلاس‌هایی که با هم تعامل دارند می‌شود. بدین ترتیب آن‌ها دیگر نیازی به دانستن جزئیات پیاده‌سازی و اجرای یکدیگر ندارند.

مثال برنامه نویسی

در این قسمت یک مثال ساده از چت روم را انجام می‌دهیم. چت روم یک mediator و کاربران حاضر در آن colleagues هستند که به یکدیگر پیام ارسال می‌کنند.
ابتدا mediator یا اتاق چت در این مثال را پیاده‌سازی می‌کنیم.

```
interface ChatRoomMediator
{
   public function showMessage(User $user, string $message);
}

// Mediator
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;
   }
}
```

سپس کاربران یا colleagues خود را داریم.

```
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);
    }
}
```

و استفاده

```
$mediator = new ChatRoom();

$john = new User('John Doe', $mediator);
$jane = new User('Jane Doe', $mediator);

$john->send('Hi there!');
$jane->send('Hey!');
// Output will be
// Feb 14, 10:58 [John]: Hi there!
// Feb 14, 10:58 [Jane]: Hey!
```

قابلیت استفاده

- استفاده مجدد

اغلب ممکن است به دلیل سلیقه توسعه دهنده، وابستگی‌ها، مدل پیاده‌سازی و...، استفاده مجدد از اجزای یک نرم‌افزار به علت ناسازگاری‌های ارتباطی با سایر اجزا در نرم‌افزارهای دیگر مشکل باشد.

هنگامی که می‌خواهید اجزای نرم‌افزار خود یا دیگران را در نرم‌افزارهای متفاوت دیگری استفاده کنید mediator مشکل شما را حل خواهد کرد. این الگو به عنوان یک شئ واسط هرج و مرج ارتباطی بین اشیاء را در خود مخفی می‌کند.
- برای پیاده‌سازی سیستم‌های مبتنی بر پیام (message-based)
- کاهش coupling

Memento

مثال در دنیای واقعی

هنگام استفاده از ماشین حساب آخرین محاسبات شما در حافظه ذخیره می‌شود (memento) تا بتوانید به آن رجوع کنید.

به بیان ساده

بازگردانی وضعیت یک شئ به حالت قبلی.
الگوی memento در مورد گرفتن و ذخیره کردن حالت جاری یک شئ است. به طوری که بتوانید آن را به آسانی بازیابی کنید. این الگو توانایی بازگرداندن یک شئ به حالت قبلی خود (بازگشت از طریق rollback) را ممکن می‌سازد.

مثال برنامه نویسی

بیایید مثال ویرایشگر متن را انجام دهیم. در این ویرایشگر حالت جاری ذخیره می‌شود که می‌توان آن را بازیابی کرد.
ابتدا یک شئ memento که محتوا را در خود نگه می‌دارد:

```
class EditorMemento
{
   protected $content;

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

```
    }

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

سپس ویرایشگر که از شئ memento استفاده می‌کند.

```
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();
    }
}
```

و استفاده

```
$editor = new Editor();
// Type some stuff
$editor->type('This is the first sentence.');
$editor->type('This is second.');
// Save the state to restore to: This is the first sentence. This is second.
$saved = $editor->save();
// Type some more
$editor->type('And this is third.');
// Output: Content before Saving
echo $editor->getContent(); // This is the first sentence. This is second.
And this is third.
// Restoring to last saved state
$editor->restore($saved);
$editor->getContent(); // This is the first sentence. This is second.
```

قابلیت استفاده

برای پیاده‌سازی عملیات undo و redo که به عنوان مثال در تراکنش‌های پایگاه داده کاربرد دارد.

Observer

مثال در دنیای واقعی

یک مثال خوب می‌تواند در مورد افراد متقاضی شغل باشد که در برخی از سایت‌های کاریابی عضو بخش اطلاع رسانی می‌شوند، تخصص‌های خود را وارد می‌کنند تا هر زمان که فرصت شغلی متناسب با تخصصشان ثبت شد به آن‌ها از طریق ایمیل اطلاع داده شود.

به بیان ساده

یک مکانیسم مشترک یا وابستگی (dependency) بین اشیاء تعریف و با استفاده از آن هر زمان که یک شئ حالت خود را تغییر داد، تمام وابستگانش را مطلع می‌کند.

در ویکی پدیا گفته شده

در الگوی observer یک شئ به نام subject، فهرست وابستگی‌هایش را می‌دارد که به آن observers گفته می‌شود و هرگونه تغییر در وضعیتش را به‌طور خودکار و معمولاً با صدا کردن یکی از متد آنها به اطلاع آن اشیا می‌رساند.
این الگو معمولاً برای پیاده‌سازی سامانه‌های توزیع‌شده رسیدگی رویداد در نرم‌افزارهای رویداد محور (event driven) استفاده می‌شود.
الگوی observer همچنین بخش مهمی در الگوی معماری مدل-نما-کنترل‌گر (MVC) است.

مثال برنامه نویسی

مثال همین بخش را انجام می‌دهیم. ما افراد متقاضی شغل داریم که باید جدیدترین آگهی‌های استخدام برایشان ارسال شود.

```
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 'Hi ' . $this->name . '! New job posted: '. $job->getTitle();
    }
}
```

کلاس کاریابی را انجام می‌دهیم که با استفاده از آن افراد متقاضی شغل در هنگام اضافه شدن فرصت شغلی جدید مطلع می‌شوند.

```
class EmploymentAgency 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);
    }
}
```

و استفاده

```
// Create subscribers
$johnDoe = new JobSeeker('John Doe');
$saraDoe = new JobSeeker('Sara Doe');

// Create publisher and attach subscribers
$jobPostings = new EmploymentAgency();
$jobPostings->attach($johnDoe);
$jobPostings->attach($saraDoe);

// Add a new job and see if subscribers get notified
$jobPostings->addJob(new JobPost('Software Engineer'));
// Output
// Hi John Doe! New job posted: Software Engineer
// Hi Sara Doe! New job posted: Software Engineer
```

قابلیت استفاده

هنگامی که نیاز است بعد از تغییر وضعیت یک شئ سایر اشیاء نیز تغییر کنند.

Visitor

مثال در دنیای واقعی

تصور کنید توریست به کشور شما سفر می‌کند. برای ورورد به یک کشور به ویزا نیاز است. پس از ورود، می‌توانند بدون نیاز به مجوز از اماکن گردشگری کشور دیدن کنند. فقط کافی است این اماکن را برای بازدید به آن‌ها نشان دهید. الگوی visitor به شما دقیقاً همین امکان را می‌دهد.

به بیان ساده

visitor به شما امکان اضافه کردن عملیات بیشتر به اشیاء بدون نیاز به اصلاح آن‌ها را می‌دهد. در این الگو الگوریتم‌ها از اشیایی که در آن عمل می‌کنند جدا می‌شوند.
به‌طور خلاصه، visitor اجازه افزودن توابع مجازی به یک خانواده از کلاس‌ها بدون تغییر خود کلاس‌ها را می‌دهد. به جای آن یک کلاس visitor ایجاد می‌شود و فعالیت مخصوص توابع مجازی را پیاده‌سازی می‌کند.

در ویکی پدیا گفته شده

الگوی طراحی visitor راهی برای جدا کردن یک الگوریتم از ساختار شئ است که روی آن عمل می‌کند. یک نتیجه عملی این جدایی توانایی برای اضافه کردن عملیات جدید به ساختارهای شئ موجود بدون تغییر ساختار آن است که یک راه برای تبعیت از اصل باز/بسته می‌باشد.

مثال برنامه نویسی

بیایید نمونه‌ای از باغ وحش را پیاده‌سازی کنیم. در آن چندین حیوان وجود دارند که باید آن‌ها را صدا کنیم.

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

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

سپس رفتار حیوانات را پیاده‌سازی می‌کنیم. مثلاً غرش شیر و...

```
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);
    }
}
```

و اکنون نوبت پیاده‌سازی visitor است.

```
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();
    }
}
```

نهایتاً استفاده از آن

```
$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!
```

حالا می‌خواهیم یک رفتار دیگر برای حیوانات پیاده کنیم. (همان عملیاتی که در مورد اضافه کردن آن بدون تغییر ساختار قبلاً صحبت شد).

```
class Jump implements AnimalOperation
{
    public function visitMonkey(Monkey $monkey)
    {
        echo 'Jumped 20 feet high! on to the tree!';
```

```
    }

    public function visitLion(Lion $lion)
    {
        echo 'Jumped 7 feet! Back on the ground!';
    }

    public function visitDolphin(Dolphin $dolphin)
    {
        echo 'Walked on water a little and disappeared';
    }
}
```

و دوباره استفاده

```
$monkey = new Monkey();
$lion = new Lion();
$dolphin = new Dolphin();
$speak = new Speak();
$jump = new Jump();

$monkey->accept($speak);// Ooh oo aa aa!
$monkey->accept($jump);// Jumped 20 feet high! on to the tree!
$lion->accept($speak);// Roaaar!
$lion->accept($jump);// Jumped 7 feet! Back on the ground!
$dolphin->accept($speak);// Tuut tutt tuutt!
$dolphin->accept($jump);// Walked on water a little and disappeared
```

قابلیت استفاده

بدون الگوی visitor می‌توانستیم با یک ساختار سلسله مراتبی وراثت برای حیوانات این کار را انجام دهیم. ساده بود اما هر زمان که نیاز به عملیات جدید داشتیم لازم بود تک تک حیوانات را اصلاح کنیم.

Strategy

مثال در دنیای واقعی

مرتب سازی data را در نظر بگیرید. ابتدا از مرتب سازی bubble sort استفاده می‌کنیم اما با افزایش آیتم‌ها این الگوریتم خیلی کند می‌شود. quick sort را جایگرین آن می‌کنیم. هر چند عمل‌کرد بهتری با مجموعه داده‌های بزرگ دارد، اما برای مجموعه داده‌های کوچک‌تر بسیار کند است. به این منظور یک استراتژی برای پیاده‌سازی در نظر می‌گیریم. روی مجموعه داده‌های کوچک bubble sort انجام شود و برای بزرگ‌ترها quick sort.

به بیان ساده

الگوی strategy این امکان را می‌دهد تا الگوریتم‌ها بر اساس موقعیت و شرایط متفاوت تغییر کنند.

در این الگو خانواده‌ای از الگوریتم‌ها را تعریف می‌کنیم و هر یک را در کلاس جداگانه و مستقل قرار می‌دهیم تا قابل انتخاب یا تعویض باشند.

در ویکی پدیا گفته شده

الگوی strategy قابلیت انتخاب یک الگوریتم در زمان اجرا را ممکن می‌سازد. این الگو:

- یک خانواده از الگوریتم‌ها را معرفی می‌کند
- هر الگوریتم را کپسوله می‌کند
- و باعث می‌شود الگوریتم‌های آن خانواده به‌طور قابل تعویض قابل استفاده باشند.

الگوی strategy یکی از الگوهای موجود در کتاب مشهور الگوهای طراحی است که توسط گاما و دیگران نوشته شده‌است و مفهوم استفاده از الگوهای طراحی برای توصیف نحوه طراحی انعطاف‌پذیر و قابل استفاده مجدد شئ گرا در نرم‌افزار را محبوب ساخت.

مثال برنامه نویسی

مثال همین قسمت را انجام می‌دهیم. ابتدا ما یک interface از strategy و strategy های متفاوت جستجو را پیاده‌سازی می‌کنیم.

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

class BubbleSortStrategy implements SortStrategy
{
   public function sort(array $dataset): array
   {
      echo "Sorting using bubble sort";
      // Do sorting
      return $dataset;
   }
}

class QuickSortStrategy implements SortStrategy
{
   public function sort(array $dataset): array
   {
      echo "Sorting using quick sort";
      // Do sorting
      return $dataset;
   }
}
```

سپس کلاس مرتب سازی که می‌تواند از strategy های جستجو استفاده کند.

```
class Sorter
{
    protected $sorter;

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

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

و استفاده

```
$dataset = [1, 5, 4, 3, 2, 8];

$sorter = new Sorter(new BubbleSortStrategy());
$sorter->sort($dataset); // Output: Sorting using bubble sort

$sorter = new Sorter(new QuickSortStrategy());
$sorter->sort($dataset); // Output: Sorting using quick sort
```

قابلیت استفاده

زمانی که کلاس‌های مشابهی دارید که فقط در رفتارهایشان متفاوت هستند.

می‌توان با استفاده از این الگو منطق تجاری یک کلاس را از جزئیات اجرای آن جدا کرد. به شرطی که این جزئیات برای آن منطق مهم نباشد.
و یا هنگامی که کلاس شما برای اجرای الگوریتم‌ها به شروط متفاوتی بستگی دارد.

State

مثال در دنیای واقعی

یک اپلیکیشن نقاشی را تصور کنید، قلمو را برای ترسیم انتخاب می‌کنید. اکنون قلمو بر اساس رنگ انتخاب شده رفتار خود را تغییر می‌دهد. یعنی اگر رنگ قرمز را انتخاب کرده باشید با رنگ قرمز نقاشی می‌کنید و اگر آبی را انتخاب کنید به رنگ آبی نقاشی انجام می‌شود.

به بیان ساده

اجازه می‌دهد یک شئ هنگام تغیر وضعیت داخلی خود رفتارش را نیز تغییر دهد.

مثال برنامه نویسی

مثالی از یک ویرایشگر متن را انجام می‌دهیم که به شما اجازه تغییر متنی که تایپ کرده‌اید را می‌دهد. مثلاً اگر شما حالت bold را انتخاب کنید نوشته‌های شما زخیم می‌شوند و...
ابتدا interface و حالت‌های مورد نظرمان را می‌نویسیم.

```
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 DefaultText implements WritingState
{
    public function write(string $words)
    {
        echo $words;
    }
}
```

سپس ویرایشگر متن را پیاده می‌کنیم.

```
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);
    }
}
```

و استفاده

```
$editor = new TextEditor(new DefaultText());

$editor->type('First line');

$editor->setState(new UpperCase());

$editor->type('Second line');
$editor->type('Third line');

$editor->setState(new LowerCase());

$editor->type('Fourth line');
$editor->type('Fifth line');

// Output:
// First line
// SECOND LINE
// THIRD LINE
// fourth line
// fifth line
```

قابلیت استفاده

وقتی رفتار شئ به وضعیت آن بستگی دارد و باید بتواند در زمان اجرا مطابق حالت جدید رفتار خود را تغییر دهد.
هنگامی که عملیات زیاد همراه با شروطی داریم که به وضعیت شئ بستگی دارد.
با استفاده از این الگو می‌توان حالت‌های جدیدی اضافه کنید و یا حالت‌های موجود را به طور مستقل از هم تغییر دهید. این موضوع هزینه نگهداری را کاهش می‌دهد.

Template Method

مثال در دنیای واقعی

فرض کنید در حال ساخت خانه هستیم و مراحل کار بدین صورت است:
- آماده سازی پایه‌های خانه
- ساخت دیوارها
- ساخت سقف
- انجام طبقه‌های دیگر

ترتیب این مراحل هرگز نمی‌تواند تغییر کند. شما نمی‌توانید اول سقف و سپس پایه‌ها را بسازید اما می‌توانید هر یک از مراحل را تغییر داخلی دهید. مثلاً دیوارها را از چوب یا آجر بسازید.

به بیان ساده

template method اسکلت انجام کار را تعریف می‌کند اما پیاده‌سازی آن را به کلاس‌های فرزند می‌سپارد.

مثال برنامه نویسی

تصور کنید ما یک ابزار برای build و test داریم.
ابتدا اسکلت کارمان را تعریف می‌کنیم.

```
abstract class Builder
```

```
{
    // Template method
    final public function build()
    {
        $this->test();
        $this->lint();
        $this->assemble();
        $this->deploy();
    }

    abstract public function test();
    abstract public function lint();
    abstract public function assemble();
    abstract public function deploy();
}
```

و سپس پیاده‌سازی‌هایی از آن را انجام می‌دهیم.

```
class AndroidBuilder extends Builder
{
    public function test()
    {
        echo 'Running android tests';
    }

    public function lint()
    {
        echo 'Linting the android code';
    }

    public function assemble()
```

```
    {
        echo 'Assembling the android build';
    }

    public function deploy()
    {
        echo 'Deploying android build to server';
    }
}

class IosBuilder extends Builder
{
    public function test()
    {
        echo 'Running ios tests';
    }

    public function lint()
    {
        echo 'Linting the ios code';
    }

    public function assemble()
    {
        echo 'Assembling the ios build';
    }

    public function deploy()
    {
        echo 'Deploying ios build to server';
    }
}
```

و استفاده

```
$androidBuilder = new AndroidBuilder();
$androidBuilder->build();

// Output:
// Running android tests
// Linting the android code
// Assembling the android build
// Deploying android build to server

$iosBuilder = new IosBuilder();
$iosBuilder->build();
// Output:
// Running ios tests
// Linting the ios code
// Assembling the ios build
// Deploying ios build to server
```

قابلیت استفاده

وقتی لازم است ساختار/اسکلت/قالب از قبل مشخص باشد و به سایرین فقط اجازه دهیم پیاده‌سازی آن را توسعه دهند.

متشکرم

از اینکه کتاب رو انتخاب و مطالعه کردین متشکرم. شما با علاقه و پشتکاری که داشتین به انتهای کتاب رسیدین. امیدوارم براتون مفید باشه.

هر چند قبل از انتشار بارها کتاب رو بررسی کردم و از دوستان نزدیکم خواستم اون رو مطالعه کنن و نظر بدن، باز هم احتمال اشتباه وجود داره. به همین خاطر اگر در این کتاب اشتباهی از من دیدین ممنون میشم اون رو اطلاع بدین (**اطلاعات تماس در ابتدای کتاب هست**).

همچنین ممکنه من کتاب رو ویرایش کنم. می‌تونید هر از گاهی به صفحهٔ کتاب در وبلاگم سری بزنید و آخرین بروزرسانی ها رو در اونجا ببینید.

http://www.developit.ir**/design-patterns**

موفق باشید.