Design Patterns Overview

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!

Intro

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

Disclaimer

Usually design patterns are applicable to “rather complex” cases, i.e., you don’t need a design pattern for “Hello world” program ☺. And also design pattern usage require at least some experience. So, here we’ll only talk briefly about the concept and see some most common patterns.

A lot of useful information

Please refer to these sources for a lot of descriptions and samples on a huge variety of design patterns in PHP:
https://refactoring.guru/design-patterns
https://designpatternsphp.readthedocs.io/en/latest/
https://phptherightway.com/pages/Design-Patterns.html

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. The most common reason for this is to control access to some shared resource, e.g., a database or a file.

More details: https://refactoring.guru/design-patterns/singleton

Singleton, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class Logger
{
    // The actual singleton's instance almost always resides inside a static field.
    private static Logger $instance;

    // This we'll use for logging.
    private $logResource;

    // Singleton's constructor should not be public.
    private function __construct()
    {
        $this->logResource = fopen('php://stdout', 'w');
    }

    // This method is used to get a Singleton's instance.
    public static function getInstance()
    {
        if (!isset(self::$instance)) {
            // Yes, we may use 'new Logger', still this is more accurate:
            self::$instance = new static();
        }
        return self::$instance;
    }

    // Main business logic :).
    public function logRecord(string $message): void
    {
        $dateAndTime = date('Y.m.d H:i:s');
        fwrite($this->logResource, $dateAndTime . ': ' . $message . "\n");
    }
}
```

```php
// This is how to get an instance of a Singleton.
$loggerOne = Logger::getInstance();

// Let's make sure there is only one instance.
$loggerTwo = Logger::getInstance();

if ($loggerOne === $loggerTwo) {
    $loggerOne->logRecord("It works!");
}
```

Singleton, big real-life-like sample

```php
<?php

class Logger
{
    // The actual singleton's instance almost always resides inside a static field.
    private static $instance;

    // This we'll use for logging.
    private $logResource;

    // Singleton's constructor should not be public.
    // It can be provate (or protected, if we want to allow inheritance).
    private function __construct() {
        $this->logResource = fopen('php://stdout', 'w');
    }

    // Cloning and unserialization are not permitted for singletons.
    protected function __clone() { }

    public function __wakeup()
    {
        throw new Exception("Cannot unserialize singleton");
    }

    // The destructor should be public.
    public function __destruct() {
        fclose($this->logResource);
    }

    // This method is used to get a Singleton's instance.
    public static function getInstance()
    {
        if (!isset(self::$instance)) {
            // Yes, we may use 'new Logger', still this is more accurate:
            self::$instance = new static();
        }
        return self::$instance;
    }

    // Main business logic :).
    public function logRecord(string $message): void
    {
        $dateAndTime = date('Y.m.d H:i:s');
        fwrite($this->logResource, $dateAndTime . ': ' . $message . "\n");
    }

    // Or we may even simplify the approach (combine object creation and business logic).
    public static function logRecordAtOnce(string $message): void
    {
        $logger = static::getInstance();
        $logger->logRecord($message);
    }

}

// This is how to get an instance of a Singleton.
$loggerOne = Logger::getInstance();

// Let's make sure there is only one instance.
$loggerTwo = Logger::getInstance();

if ($loggerOne === $loggerTwo) {
    $loggerOne->logRecord("It works!");
}

// Simplified approach.
Logger::logRecordAtOnce("Done!");
```

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. The Builder suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

More details: https://refactoring.guru/design-patterns/builder

Builder, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class BuildablePictureElement
{
    private string $src;
    private string $alt;
    private int $width;
    private int $height;

    // All setters should return the object itself
    // to allow chaining: $img->setSrs(...)->setAlt(...)

    public function setSrc(string $src): BuildablePictureElement
    {
        $this->src = $src;
        return $this;
    }

    public function setAlt(string $alt): BuildablePictureElement
    {
        $this->alt = $alt;
        return $this;
    }

    public function setWidth(int $width): BuildablePictureElement
    {
        $this->width = $width;
        return $this;
    }

    public function setHeight(int $height): BuildablePictureElement
    {
        $this->height = $height;
        return $this;
    }

    public function getImg(): string
    {
        return '<img src="' . $this->src . '" alt="' . $this->alt . '" width="' . $this->width . '" height="' . $this->height . '">';
    }

}
```

```php
echo $img = (new BuildablePictureElement())
    ->setSrc('1.jpg')
    ->setAlt('Picture')
    ->setHeight(10)
    ->setWidth(20)
    ->getImg();
```

Builder, big real-life-like sample

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The Factory Method pattern suggests that you replace direct object construction calls (using the *new* operator) with calls to a special factory method. The objects are still created via the *new* operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as *products*.

More details: https://refactoring.guru/design-patterns/factory-method

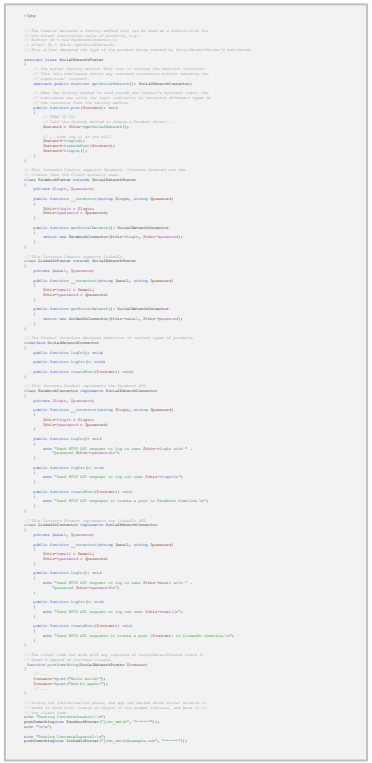
Factory Method, “almost wrong” (because of extra-simplicity) sample

```php
<?php

abstract class Writer
{
    public const COLOR_BLUE = 1;
    public const COLOR_RED = 2;
    abstract protected function
    getWritingTool(?string $color =
     self::COLOR_BLUE): WritingTool;
}

abstract class WritingTool
{

}

class BluePen extends WritingTool
{
    public function write()
    {
        echo "BluePen\n";
    }

}

class RedPen extends WritingTool
{
    public function write()
    {
        echo "RedPen\n";
    }

}
```

```php
class Pencil extends WritingTool
{
    public function write()
    {
        echo "Pencil\n";
    }
}

class PenWriter extends Writer
{
    public function write(?string $color = Writer::COLOR_BLUE): void
    {
        $writingTool = $this->getWritingTool($color);
        $writingTool->write();
    }

    protected function getWritingTool(?string $color =
                        Writer::COLOR_BLUE): WritingTool
    {
        if ($color == Writer::COLOR_BLUE) {
            return new BluePen();
        } else {
            return new RedPen();
        }
    }
}
```

```php
class PencilWriter extends Writer
{
    public function write(): void
    {
        $writingTool = $this->getWritingTool();
        $writingTool->write();
    }

    protected function getWritingTool(?string $color =
                         Writer::COLOR_BLUE): WritingTool
    {
        return new Pencil();
    }
}

$penWriter = new PenWriter();
$pencilWriter = new PencilWriter();
$penWriter->write(Writer::COLOR_BLUE);
$penWriter->write(Writer::COLOR_RED);
$pencilWriter->write(null);
```

Factory Method, big real-life-like sample



This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes. The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family. Then you can make all variants of products follow those interfaces.

More details: https://refactoring.guru/design-patterns/abstract-factory

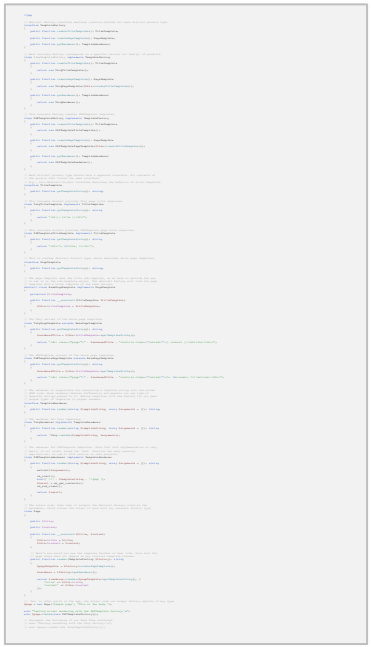
Abstract Factory, “almost wrong” (because of extra-simplicity) sample

```php
<?php

abstract class WritingFactory
{
    abstract public function createTool(): Tool;

    abstract public function createSurface(): Surface;
}

class OfficeWriting extends WritingFactory
{

    public function createTool(): Tool
    {
        return new Pen;
    }

    public function createSurface(): Surface
    {
        return new Plastic;
    }

}

class HomeWriting extends WritingFactory
{

    public function createTool(): Tool
    {
        return new Pencil;
    }

    public function createSurface(): Surface
    {
        return new Paper;
    }
}
```

```php
abstract class Tool
{
}

class Pen extends Tool
{
    public function write()
    {
        return "Pen";
    }
}

class Pencil extends Tool
{
    public function write()
    {
        return "Pencil";
    }
}

abstract class Surface
{
}

class Paper extends Surface
{
    public function writeWith(Tool $tool)
    {
        echo "Writing on Paper with " .
             $tool->write() . "\n";
    }
}
```

```php
class Plastic extends Surface
{
    public function writeWith(Tool $tool)
    {
        echo "Writing on Plastic with " . $tool->write() . "\n";
    }
}

class CreativityProcess
{
    private $tool;
    private $surface;

    public function __construct(WritingFactory $writingFactory)
    {
        $this->tool = $writingFactory->createTool();
        $this->surface = $writingFactory->createSurface();
    }

    public function write() {
        $this->surface->writeWith($this->tool);
    }
}

$officeCreativity = new CreativityProcess(new OfficeWriting());
$officeCreativity->write();

$homeCreativity = new CreativityProcess(new HomeWriting());
$homeCreativity->write();
```

Abstract Factory, big real-life-like sample



This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

More details: https://refactoring.guru/design-patterns/adapter

Adapter, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class Clock
{
    public function getTime() : int
    {
        return time();
    }
}

class HumanClock extends Clock
{

    public function getHumanTime() : string
    {
        return date('H:i:s', $this->getTime());
    }

}

$originalClock = new Clock();
echo $originalClock->getTime() . "\n";

$wrappedClock = new HumanClock();
echo $wrappedClock->getHumanTime() . "\n";
```

Adapter, big real-life-like sample

```php
<?php

// The Target interface represents the interface that your application's classes already follow.
interface Notification
{
    public function send(string $title, string $message);
}

// Here's an example of the existing class that follows the Target interface.
class EmailNotification implements Notification
{
    private $adminEmail;

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

    public function send(string $title, string $message): void
    {
        // Uncomment, if you want real e-mails to be sent.
        // mail($this->adminEmail, $title, $message);
        echo "Sent email with title '$title' to '{$this->adminEmail}' that says '$message'.";
    }
}

// The Adaptee is some useful class, incompatible with the Target interface. You
// can't just go in and change the code of the class to follow the Target
// interface, since the code might be provided by a 3rd-party library.
class SlackApi
{
    private $login;
    private $apiKey;

    public function __construct(string $login, string $apiKey)
    {
        $this->login = $login;
        $this->apiKey = $apiKey;
    }

    public function login(): void
    {
        // Send authentication request to Slack web service.
        echo "Logged in to a slack account '{$this->login}'.\n";
    }

    public function sendMessage(string $chatId, string $message): void
    {
        // Send message post request to Slack web service.
        echo "Posted following message into the '$chatId' chat: '$message'.\n";
    }
}

// The Adapter is a class that links the Target interface and the Adaptee class.
// In this case, it allows the application to send notifications using Slack API.
class SlackNotification implements Notification
{
    private $slack;
    private $chatId;

    public function __construct(SlackApi $slack, string $chatId)
    {
        $this->slack = $slack;
        $this->chatId = $chatId;
    }

    // An Adapter is not only capable of adapting interfaces, but it can also
    // convert incoming data to the format required by the Adaptee.
    public function send(string $title, string $message): void
    {
        $slackMessage = "*" . $title . "* " . strip_tags($message);
        $this->slack->login();
        $this->slack->sendMessage($this->chatId, $slackMessage);
    }
}

// The client code can work with any class that follows the Target interface.
function someSomething(Notification $notification)
{
    // ...

    echo $notification->send("Website is down!",
        "<strong style="color:red;font-size: 50px;">Alert!</strong> " .
        "Our website is not responding. Call admins and bring it up!");

    // ...
}

echo "Client code is designed correctly and works with email notifications:\n";
$notification = new EmailNotification("developers@example.com");
someSomething($notification);
echo "\n\n";


echo "The same client code can work with other classes via adapter:\n";
$slackApi = new SlackApi("example.com", "XXXXXXXX");
$notification = new SlackNotification($slackApi, "Example.com Developers");
someSomething($notification);
```

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Composite

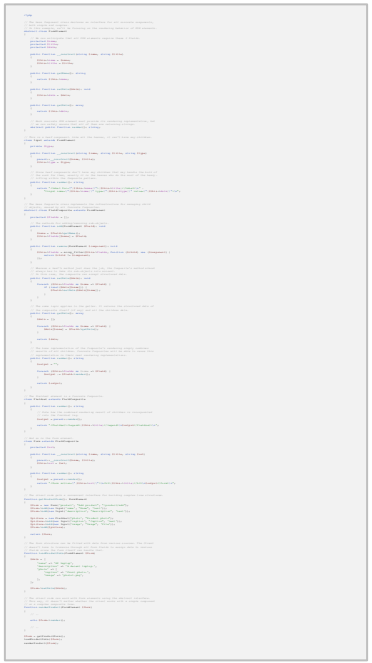
Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects. Using the Composite pattern makes sense only when the core model of your app can be represented as a tree. For example, imagine that you have two types of objects: *Products* and *Boxes*. A Box can contain several *Products* as well as a number of smaller *Boxes*. These little *Boxes* can also hold some *Products* or even smaller *Boxes*, and so on. The Composite pattern suggests that you work with *Products* and *Boxes* through a common interface which declares a method for calculating the total price.

More details: https://refactoring.guru/design-patterns/composite

Composite, “almost wrong” (because of extra-simplicity) sample

```php
<?php

// Disclaimer! This is NOT a tree-builder or math formula builder!
// It does NOT follow any specific rules.
// It just shows the idea of "composing a component with another components"!
class BracketedExpression
{
    protected $subexpressions = [];

    public function __construct(string|BracketedExpression $subexpression)
    {
        $this->subexpressions[] = $subexpression;
    }

    public function add(string|BracketedExpression $subexpression): void
    {
        $this->subexpressions[] = $subexpression;
    }

    public function render(): string
    {
        $output = "";

        foreach ($this->subexpressions as $subexpression) {
            if ($subexpression instanceof BracketedExpression) {
                $output .= '{' . $subexpression->render() . '}';
            } else {
                $output = '{' . $output . $subexpression . '}';
            }

        }

        return $output;
    }
}

$bracketedExpression = new BracketedExpression('A,B');
$bracketedExpression->add('X');
$bracketedExpression->add(new BracketedExpression('C,D,E'));
echo $bracketedExpression->render();
```

Composite, big real-life-like sample



This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object. The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

More details: https://refactoring.guru/design-patterns/proxy

Proxy, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class FileWriter
{
    private $fileResource;

    public function openFile(string $fileName): void
    {
        $this->fileResource = fopen($fileName, 'wb');
    }

    public function writeToFile(string $data): void
    {
        fputs($this->fileResource, $data);
    }

    public function closeFile(): void
    {
        fclose($this->fileResource);
    }
}
```

```php
class ProxiedFileWriter
{
    private FileWriter $fileWriter;
    private string $dataToWrite = '';

    public function openFile(string $fileName): void
    {
        $this->fileWriter = new FileWriter();
        $this->fileWriter->openFile($fileName);
    }

    public function writeToFile(string $data): void
    {
        $this->dataToWrite .= $data;
        if (strlen($this->dataToWrite) < 10) {
            echo "Too few data. Add more.\n";
        } else {
            $this->fileWriter->writeToFile($this->dataToWrite);
            $this->dataToWrite = '';
            echo "\nWritten!\n";
        }
    }

    public function closeFile(): void
    {
        $this->fileWriter->closeFile();
    }
}
```

```php
$proxiedFileWriter = new ProxiedFileWriter();
$proxiedFileWriter->openFile('php://stdout');
$proxiedFileWriter->writeToFile('ABC');
$proxiedFileWriter->writeToFile('DEFG');
$proxiedFileWriter->writeToFile('HIJK');
```

Proxy, big real-life-like sample

```php
<?php

// The Subject interface describes the interface of a real object.
interface Downloader
{
    public function download(string $url): string;
}

// The Real Subject does the real job, albeit not in the most efficient way.
// When a client tries to download the same file for the second time, our
// downloader does just that, instead of fetching the result from cache.
class SimpleDownloader implements Downloader
{
    public function download(string $url): string
    {
        echo "Downloading a file from the Internet.\n";
        $result = file_get_contents($url);
        echo "Downloaded bytes: " . strlen($result) . "\n";

        return $result;
    }
}

// The Proxy class is our attempt to make the download more efficient. It wraps
// the real downloader object and delegates it the first download calls. The
// result is then cached, making subsequent calls return an existing file
// instead of downloading it again.
// Note that the Proxy MUST implement the same interface as the Real Subject.
class CachingDownloader implements Downloader
{
    private $downloader;
    private $cache = [];

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

    public function download(string $url): string
    {
        if (!isset($this->cache[$url])) {
            echo "CacheProxy MISS. ";
            $result = $this->downloader->download($url);
            $this->cache[$url] = $result;
        } else {
            echo "CacheProxy HIT. Retrieving result from cache.\n";
        }
        return $this->cache[$url];
    }
}

// The client code may issue several similar download requests. In this case,
// the caching proxy saves time and traffic by serving results from cache.
// The client is unaware that it works with a proxy because it works with
// downloaders via the abstract interface.
function sampleOfProxyUsage(Downloader $subject)
{
    // ...

    $result = $subject->download("http://example.com/");

    // Duplicate download requests could be cached for a speed gain.

    $result = $subject->download("http://example.com/");

    // ...
}

echo "Executing client code with real subject:\n";
$realSubject = new SimpleDownloader();
sampleOfProxyUsage($realSubject);

echo "\n";

echo "Executing the same client code with a proxy:\n";
$proxy = new CachingDownloader($realSubject);
sampleOfProxyUsage($proxy);
```

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

More details: https://refactoring.guru/design-patterns/iterator

Iterator, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class StringIterator implements Iterator
{
    private string $stringData;
    private int $stringPosition;

    public function __construct(string $stringData)
    {
        $this->stringData = $stringData;
        $this->stringPosition = 0;
    }

    public function current(): mixed
    {
        if ($this->stringPosition < strlen($this->stringData)) {
            return $this->stringData[$this->stringPosition];
        } else {
            return null;
        }
    }

    public function next(): void
    {
        $this->stringPosition++;
    }

    public function key(): mixed
    {
        return $this->stringPosition;
    }

    public function valid(): bool
    {
        if ($this->stringPosition < strlen($this->stringData)) {
            return true;
        } else {
            return false;
        }
    }

    public function rewind(): void
    {
        $this->stringPosition = 0;
    }
}

$stringIterator = new StringIterator('ABCDE');
foreach ($stringIterator as $position => $letter) {
    echo "[" . $position . "] = [" . $letter . "]\n";
}
```

Iterator, big real-life-like sample

```php
<?php

// CSV File Iterator (by Josh Lockhart).
class CsvIterator implements Iterator
{
    const ROW_SIZE = 4096;

    protected $filePointer = null;

    protected $currentElement = null;

    protected $rowCounter = null;

    protected $delimiter = null;

    public function __construct($file, $delimiter = ',')
    {
        try {
            $this->filePointer = fopen($file, 'rb');
            $this->delimiter = $delimiter;
        } catch (\Exception $e) {
            throw new \Exception('The file "' . $file . '" cannot be
read.');
        }
    }

    public function rewind(): void
    {
        $this->rowCounter = 0;
        rewind($this->filePointer);
    }

    // This method returns the current CSV row as a 2-dimensional
array.
    public function current(): array
    {
        $this->currentElement = fgetcsv($this->filePointer,
self::ROW_SIZE, $this->delimiter);
        $this->rowCounter++;

        return $this->currentElement;
    }

    // This method returns the current row number.
    public function key(): int
    {
        return $this->rowCounter;
    }

    // This method checks if the end of file has been reached.
    public function next(): bool
    {
        if (is_resource($this->filePointer)) {
            return !feof($this->filePointer);
        }

        return false;
    }

    // This method checks if the next row is a valid row.
    public function valid(): bool
    {
        if (!$this->next()) {
            if (is_resource($this->filePointer)) {
                fclose($this->filePointer);
            }

            return false;
        }

        return true;
    }
}

$csv = new CsvIterator(__DIR__ . '/cats.csv');

foreach ($csv as $key => $row) {
    print_r($row);
}
```

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

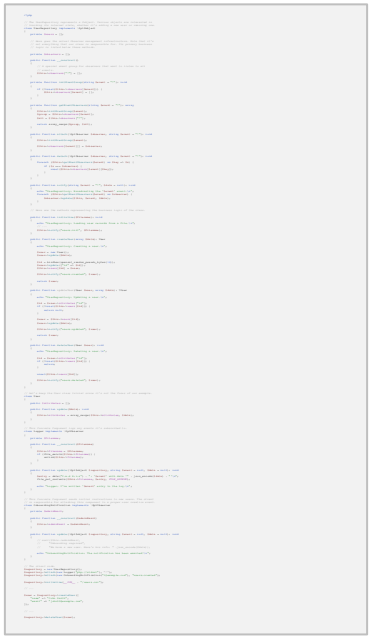
More details: https://refactoring.guru/design-patterns/observer

Observer, “almost wrong” (because of extra-simplicity) sample

```php
<?php

class Actor
{
    private $observers = [];

    public function performAction(string $action): void
    {
        echo "Actor: I'm doing " . $action . "!\n";
        $this->notifyObservers($action);
    }

    public function addObserver(Observer $observer)
    {
        $this->observers[] = $observer;
    }

    private function notifyObservers(string $action)
    {
        foreach ($this->observers as $observer) {
            $observer->reactToObservableAction($action);
        }
    }

}
```

```php
abstract class Observer
{
    abstract public function reactToObservableAction(string $action);
}

class CuriousObserver extends Observer
{
    public function reactToObservableAction(string $action)
    {
        echo "Hmm... I'm curious! Nice " . $action . "!\n";
    }
}

class AttentiveObserver extends Observer
{
    public function reactToObservableAction(string $action)
    {
        echo "Hmm... I'm attentive! Nice " . $action . "!\n";
    }
}

$actor = new Actor();
$curiousObserver = new CuriousObserver();
$attentiveObserver = new AttentiveObserver();

$actor->addObserver($curiousObserver);
$actor->addObserver($attentiveObserver);

$actor->performAction('something');
```

Observer, big real-life-like sample



This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*. The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

More details: https://refactoring.guru/design-patterns/strategy

Strategy, “almost wrong” (because of extra-simplicity) sample

```php
<?php

abstract class BeautifulTextMaker
{
    abstract public function beautifyText(string $text) : string;
}

class BoldTextMaker extends BeautifulTextMaker
{
    public function beautifyText(string $text) : string
    {
        return '<b>' . $text . '</b>';
    }
}

class ItalicTextMaker extends BeautifulTextMaker
{
    public function beautifyText(string $text) : string
    {
        return '<i>' . $text . '</i>';
    }
}

class UnderlinedTextMaker extends BeautifulTextMaker
{
    public function beautifyText(string $text) : string
    {
        return '<u>' . $text . '</u>';
    }
}
```

```php
class TextProcessor
{
    private BeautifulTextMaker $beautifulTextMaker;

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

    public function processText(string $text)
    {
        return $this->beautifulTextMaker->beautifyText($text);
    }

}


$boldTextProcessor = new TextProcessor(new BoldTextMaker);
echo $boldTextProcessor->processText('Bold') . "\n";

$italicTextProcessor = new TextProcessor(new ItalicTextMaker);
echo $italicTextProcessor->processText('Italic') . "\n";

$underlinedTextProcessor = new TextProcessor(new UnderlinedTextMaker);
echo $underlinedTextProcessor->processText('Underlined') . "\n";
```

Strategy, big real-life-like sample

This code here is just for the reference (in case you don't have the handouts; otherwise – see the file in the handouts ☺).

Afterword

There is no point attempting to memorize design patterns. It's usually useless. Just read (and re-read) pattern descriptions from time to time. And there is a huge chance that you'll remember the idea when you need it. And you may always find concrete implementation details in a variety of manuals.

Once again, here's a lot of useful information:
https://refactoring.guru/design-patterns
https://designpatternsphp.readthedocs.io/en/latest/
https://phptherightway.com/pages/Design-Patterns.html

P.S. More ideas to use

In order to understand best programming practices better, read these articles:

- SOLID: https://accesto.com/blog/solid-php-solid-principles-in-php/
- KISS: https://thevaluable.dev/kiss-principle-explained/
- DRY: https://thevaluable.dev/dry-principle-cost-benefit-example/
- YAGNI: https://dev.to/gonedark/practicing-yagni---show-me-the-code-pjn
- GRASP: https://wp-punk.com/take-responsibility-into-your-control-with-grasp-principles/

These ideas are not “PHP-bound”, they will help you with programming in general. And, of course, with PHP programming ☺.

Design Patterns Overview

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!