PHP

The Right Way

LAST UPDATED: 2023-01-15 19:37:40 +0000

SHARE ON TWITTER

Design Patterns

There are numerous ways to structure the code and project for your web application, and you can put as much or as little thought as you like into architecting. But it is usually a good idea to follow common patterns because it will make your code easier to manage and easier for others to understand.

- Architectural pattern on Wikipedia
- Software design pattern on Wikipedia
- Collection of implementation examples

Factory

One of the most commonly used design patterns is the factory pattern. In this pattern, a class simply creates the object you want to use. Consider the following example of the factory pattern:

```
<?php
class Automobile
{
    private $vehicleMake;
    private $vehicleModel;

    public function __construct($make, $model)
    {
        $this->vehicleMake = $make;
        $this->vehicleModel = $model;
    }

    public function getMakeAndModel()
```

```
    {
        return $this->vehicleMake . ' ' . $this->vehicleModel;
    }
}

class AutomobileFactory
{
    public static function create($make, $model)
    {
        return new Automobile($make, $model);
    }
}

// have the factory create the Automobile object
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');

print_r($veyron->getMakeAndModel()); // outputs "Bugatti Veyron
```

This code uses a factory to create the Automobile object. There are two possible benefits to building your code this way; the first is that if you need to change, rename, or replace the Automobile class later on you can do so and you will only have to modify the code in the factory, instead of every place in your project that uses the Automobile class. The second possible benefit is that, if creating the object is a complicated job, you can do all of the work in the factory instead of repeating it every time you want to create a new instance.

Using the factory pattern isn't always necessary (or wise). The example code used here is so simple that a factory would simply be adding unneeded complexity. However if you are making a fairly large or complex project you may save yourself a lot of trouble down the road by using factories.

- Factory pattern on Wikipedia

Singleton

When designing web applications, it often makes sense conceptually and architecturally to allow access to one and only one instance of a particular class. The singleton pattern enables us to do this.

TODO: NEED NEW SINGLETON CODE EXAMPLE

The code above implements the singleton pattern using a *static* variable and the static creation method `getInstance()`. Note the following:

- The constructor `__construct()` is declared as protected to prevent creating a new instance outside of the class via the `new` operator.
- The magic method `__clone()` is declared as private to prevent cloning of an instance of the class via the `clone` operator.
- The magic method `__wakeup()` is declared as private to prevent unserializing of an instance of the class via the global function `unserialize()` .
- A new instance is created via late static binding in the static creation method `getInstance()` with the keyword `static`. This allows the subclassing of the class `Singleton` in the example.

The singleton pattern is useful when we need to make sure we only have a single instance of a class for the entire request lifecycle in a web application. This typically occurs when we have global objects (such as a Configuration class) or a shared resource (such as an event queue).

You should be wary when using the singleton pattern, as by its very nature it introduces global state into your application, reducing testability. In most cases, dependency injection can (and should) be used in place of a singleton class. Using dependency injection means that we do not introduce unnecessary coupling into the design of our application, as the object using the shared or global resource requires no knowledge of a concretely defined class.

- Singleton pattern on Wikipedia

Strategy

With the strategy pattern you encapsulate specific families of algorithms allowing the client class responsible for instantiating a particular algorithm to have no knowledge of the actual implementation. There are several variations on the strategy pattern, the simplest of which is outlined below:

This first code snippet outlines a family of algorithms; you may want a serialized array, some JSON or maybe just an array of data:

```
<?php

interface OutputInterface
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
{
    public function load()
```

```
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

By encapsulating the above algorithms you are making it nice and clear in your code that other developers can easily add new output types without affecting the client code.

You will see how each concrete 'output' class implements an OutputInterface - this serves two purposes, primarily it provides a simple contract which must be obeyed by any new concrete implementations. Secondly by implementing a common interface you will see in the next section that you can now utilise Type Hinting to ensure that the client which is utilising these behaviours is of the correct type, in this case 'OutputInterface'.

The next snippet of code outlines how a calling client class might use one of these algorithms and even better set the behaviour required at runtime:

```
<?php
class SomeClient
{
    private $output;

    public function setOutput(OutputInterface $outputType)
    {
        $this->output = $outputType;
    }
```

```
    public function loadOutput()
    {
        return $this->output->load();
    }
}
```

The calling client class above has a private property which must be set at runtime and be of type 'OutputInterface'. Once this property is set a call to loadOutput() will call the load() method in the concrete class of the output type that has been set.

```
<?php
$client = new SomeClient();

// Want an array?
$client->setOutput(new ArrayOutput());
$data = $client->loadOutput();

// Want some JSON?
$client->setOutput(new JsonStringOutput());
$data = $client->loadOutput();
```

- Strategy pattern on Wikipedia

Front Controller

The front controller pattern is where you have a single entrance point for your web application (e.g. index.php) that handles all of the requests. This code is responsible for loading all of the dependencies, processing the request and sending the response to the browser. The front controller pattern can be beneficial because it encourages modular code and gives you a central place to hook in code that should be run for every request (such as input sanitization).

- Front Controller pattern on Wikipedia

Model-View-Controller

The model-view-controller (MVC) pattern and its relatives HMVC and MVVM let you break up code into logical objects that serve very specific purposes. Models serve as a data access layer where data is fetched and returned in formats usable throughout your application. Controllers handle the request, process the data returned from models and load views to send in the response. And views are display templates (markup, xml, etc) that are sent in the response to the web browser.

MVC is the most common architectural pattern used in the popular PHP frameworks.

Learn more about MVC and its relatives:

- MVC
- HMVC
- MVVM

Created and maintained by

Josh Lockhart
Phil Sturgeon
Project Contributors