PHP OOP WAY

Learn the art of developing maintainable OOP applications

SERGEY ZHUK

PHP OOP Way

Sergey Zhuk

This book is for sale at http://leanpub.com/phpoopway

This version was published on 2021-03-26

Contents

Dependency Injection

Now, when we know enough about OOP fundamentals, it's time to move on to the concept of the dependency injection. In an object-oriented application, objects constantly interact with each other, either by calling methods and receiving information from another object, or changing the state of the objects. In any case, objects often *depend* on each other.

Consider this simple `QueryBuilder` class. We have found that some of our queries to a database are very slow and we need to log them. So, in our query builder, we need a logger object to log requires:

```
class QueryBuilder
{
    public function execute($sql, $params)
    {
        // ...
        $logger = new Logger();
        $logger->info(
            'DB:'.$sql.';'.implode(',', $params)
        );
    }
}
```

As a quick solution - nice. Who cares? But. Something bad happens. One of our colleagues has changed `Logger` class constructor and added a required `$filename` parameter to it. And suddenly our application dies. All database queries are broken. One small change crashes the whole application. Very sad scenario. Of course, modifying a class constructor in a production code is already a strong sign of poor application design. But, bad things happen.

Hardcoded dependencies

It's time to examine our mistakes. Every time we use one class name inside another class, we couple them together. It is OK to depend on an abstract class, but you should not depend on any concrete implementation (it will be discussed in details in Dependency Inversion Principle section). The most danger thing is to instantiate an object, where it should not be created. You can argue, that how we can determine where is the right place for a particular object to be created? Let's go from the opposite, and try to find wrong places to create objects. As we have already seen in the previous example, is was not a great idea to instantiate `Logger` object right in place, where it will be used. When suddenly its constructor signature has changed we have to search through the entire code base and fix every creation of the logger. With modern IDEs it may be not so overwhelming. But we code in PHP, a dynamically typed language. Your IDE is useless with statements like this:

```
function makeInstance($class) {
    return new $class;
}

$className = 'Logger';
$logger = makeInstance($classname);
```

We can use a variable, that stores a class name to create an object of this class. You will find this only when something will be broken. So, to fix this issue we should remove the instantiation of the logger outside and pass it as an argument.

```
class QueryBuilder
{
    public function execute($sql, $params, Logger $logger = null)
    {
        // ...
        if ($logger) {
            $logger->info(
                'DB:'.$sql.';'.implode(','. $params)
            );
        }
    }
}
```

We can make this argument optional and also we should typehint it, to ensure that we can safely call `info` method on it. What improvements do we get here?

- We don't depend on constructor signature of the `Logger` class

- We don't depend on any concrete implementation. We can extend `Logger` with `InMemoryLogger` and safely use it.
- We can mock a logger for testing purposes.

When we pass a dependency as an argument to a method/constructor we *inject* it. This technique is called *dependency injection*. Instead of hardcoding our dependencies, we inject them into the object, that uses them. The *dependent* class is **never** a good place *to create a dependency* because otherwise it will violate Single Responsibility Principle. If we have a class that creates a dependency and then uses it, then there are at least two reasons this class may have to change.

Consider one more example. `Delivery` class that sends orders to a delivery system API. It has an HTTP client that is used to make HTTP requests to API endpoints.

```
class Delivery
{
    private $client;

    public function __construct()
    {
        $this->client = new HttpClient();
    }

    public function send(Order $order)
    {
        $response = $this->client
            ->post(
                '/orders/create', $order->toJson()
            );

        return json_decode($response, true);
    }

    public function getStatus($orderId)
    {
        $response = $this->client
            ->get(
                '/orders/info', $orderId
            );

        return json_decode($response, true);
    }
}
```

Again we have a hardcoded dependency of `HttpClient` that is used in every method. We have already learned that we should pass it as an argument instead of being created inside a constructor. But this object is used in every single method. It will be not very wise to add one more argument to every method of the `Delivery` class. It is much better to pass it to the constructor, assign it to a property and later any method can use it. Here is a rule of thumb: *if an object cannot be used without a dependency, this dependency should be passed as a constructor argument.* And here is the correct version of `Delivery` class with dependency injection applied:

```
class Delivery
{
    private $client;

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

    public function send(Order $order)
    {
        // ...
    }

    public function getStatus($orderId)
    {
        // ...
    }
}
```

Constructor Or Setter Injection

In the previous example, we have used a constructor injection to pass `HttpClient` dependency to the `Delivery` class. But why should we use a constructor to inject dependencies? Maybe it is better to use `setClient` method, and configure object after creation. Looks flexible, that we can keep the constructor tidier and reconfigure our object later with a new dependency using a method call. So, let's remove the constructor and use `setClient` setter to supply the dependency and see what happens.

```
class Delivery
{
    private $client;

    public function setClient(HttpClient $client)
    {
        $this->client = $client;
    }

    public function send(Order $order)
    {
        $response = $this->client
            ->post(
                '/orders/create', $order->toJson()
            );

        return json_decode($response, true);
    }

    public function getStatus($orderId)
    {
        $response = $this->client
            ->get('/orders/info', $orderId);

        return json_decode($response, true);
    }
}
```

When we remove the constructor and leave only a setter for the dependency we immediately end up with an anti-pattern. We have a successfully created object, but it still should be configured, before we can start using it. Consider this example:

```php
class DeliveryController
{
    private $delivery;

    public function __constructor(Delivery $delivery)
    {
        $this->delivery = $delivery;
    }

    public function sendOrder($orderId)
    {
        $order = Order::find($orderId);

        $delivery->sendOrder($order);
    }
}

$delivery = new Delivery();
$controller = new DeliveryControler($delivery);

$controller->sendOrder(111);
// This will error, because an object
// is not configured and cannot be used
```

`DeliveryControler` has been given a successfully created instance of the `Delivery`. But `DeliveryControler` cannot make the assumption that `Delivery` object is fully configured because there is no way to know whether `setClient` method has been called before or not. It is an ambiguity here because we have broken encapsulation in the `Delivery` class. The client code shouldn't care about the internal dependencies of the objects it uses. We should avoid such incomplete objects in the application, they cause bugs which are very difficult to find out and test. In the tests, a mock for `Delivery` class will work perfectly, but in production, `DeliveryControler` will be broken. We can avoid such incomplete objects and problems related to them by using constructor injection instead. That is why *if an object cannot be used without a dependency, this dependency should be always passed as a constructor argument*

In all other scenarios, we can safely use setter injections. We can rewrite `QueryBuilder` and instead of optional parameter for the dependency create a setter for it like this:

```
class QueryBuilder
{
    private $logger;

    public function setLogger(Logger $logger)
    {
        $this->logger = $logger;
    }

    public function execute($sql, $params)
    {
        // ...
        if ($this->logger) {
            $this->logger->info(
                'DB:'.$sql.';'.implode(','. $params)
            );
        }
    }
}
```

In this example, the dependency is passed as an argument to the setter, that requires it. Now there will be no unexpected side effects because the dependency is not encapsulated in `QueryBuilder` class (the class internal state doesn't depend on the dependency). When we provide a setter for a dependency, the client code can reconfigure an object with a new dependency. It doesn't make sense if the dependency only extends the object's functionality like for example, a logger does. But when it changes the internal object state, we can easily break encapsulation. The rule of thumb for setter injection is: *use setter injections for dependencies that are not required for an object. These dependencies should not replace the internal object's functionality, instead, they should extend it.*

Dependency Container

And now you probably have a question. Where should we create all these dependencies? It's clear that we should pass them as arguments, but we need to create them before. When we replace the instantiation of the logger with something like this, we really improve nothing:

```
$queryBuilder->query($sql, $params, new Logger());
```

You will run into the same problems as if the creation were inside the method. It looks like we need some place, where we can get a required dependency. We should not create a new instance of the logger every time we meet a method that requires it. The more *new* keywords we have in our application, the more potential problems we can face. And here dependency injection containers enter the game.

We know that a class should not configure its dependencies by itself, they should be injected into a class. But what if these dependencies have their own dependencies and you should pass them through the constructor. So where do we create all these dependencies in our application? It will be weird to create all these objects every time when we need them. And it can be a real pain to manage all of these dependencies. We need some sort of a fabric for them. An object that knows how to create, configure and prepare other objects and manage their dependencies. How does it know this? We give all of these instructions to it. It remembers them and then uses these instructions to manage the dependencies for us.

In our example with `Delivery` class, we have a constructor dependency of `HttpClient`. And `HttpClient` itself should be configured with a base URL when being constructed. So, every time when we need an object of `Delivery` class we need to do something like this:

```
$httpClient = new HttpClient('http://some-api.com');
$delivery = new Delivery($httpClient);
```

It already looks complicated. But what if we decide to log some responses from the API? We need a logger, but we know that we should inject it, instead of creating it right inside `Delivery` class.

```
$logger = new Logger();
$httpClient = new HttpClient('http://some-api.com')
$delivery = new Delivery($httpClient, $logger);
```

Things are getting more and more complex and the code looks more and more ugly. The concept behind dependency injection container is to hide from us this creation logic. We write it once and forget about it. When we need a dependency we ask a container for it and now it is a container job to create and prepare the required objects. The objects themselves do not know that they are created by a container and know nothing about it.

```php
class Container
{
    public function makeDelivery()
    {
        $logger = new Logger();

        $httpClient = new HttpClient(
            'http://some-api.com'
        );

        $delivery = new Delivery(
            $httpClient, $logger
        );

        return $delivery;
    }
}

$container = new Container();
$delivery = $container->makeDelivery();
```

The code above is a very simple example, but the main idea here is to isolate this instantiation logic from the client code and keep it in one place, so the next time when something in our dependencies changes, we need to make this change only in one place in our codebase. Our client code knows nothing about how `Delivery` class should be configured and constructed. Client code only asks a container for an object.

Of course, you shouldn't write dependency container by yourself, except for the purpose of learning. There are many good implementations in PHP community:

- PHP-DI[1]
- Pimple[2]
- Symfony The Dependency Injection Component[3]

You can choose one that you like and use it. Many of them have such powerful tools as automatic resolution via PHP Reflection API, which can be an incredibly useful thing.

Use dependency injection container when you need to manage the composition of complex objects. Instead of having to remember how to build these complex objects with all their dependencies, we can declare those rules in one place and then later we can simply ask a container to build an object for use. Modern containers can also be very useful when you need to swap different implementations.

[1] http://php-di.org
[2] http://pimple.sensiolabs.org
[3] http://symfony.com/doc/current/components/dependency_injection.html

Dependency Injection Smells

We have touched all the aspects of using Dependency Injection, from passing as arguments to resolving out of the container. Now, it's time to pay attention to some examples of *how not to use* Dependency Injection.

Too Many Dependencies

When a class has a lot of dependencies, often in a constructor it is the first sign that this has does too much and has a lot of responsibilities (violates Single Responsibility Principle):

```
class OrderController
{
    public function __construct(
        OrdersRepository $ordersRepo,
        PaymentGateway $payments,
        ShippingService $shipping,
        Logger $logger
        Email $mailer
    ) {
        $this->ordersRepo = $ordersRepo;
        $this->paymentGateway = $payments;
        $this->shipping = $shipping;
        $this->logger = $logger;
        $this->mail = $mailer;
    }
}
```

At first, examine your constructor dependencies, maybe some of them are not required to be in a constructor and can be passed via method injection. For example, it is unlikely that we will need an instance of `ShippingService` to process payment. But this approach does not really solve the problem.

Another option will be to pass only a container to the constructor and later resolve all of these dependencies.

```
class OrderController
{
    public function __construct(Container $container)
    {
        $this->ordersRepo = $container->get('ordersRepository');

        $this->paymentGateway = $container->get('paymentGateway');

        $this->shipping = $container->get('PaymentGateway');

        $this->logger = $container->get('Logger');

        $this->mail = $container->get('Email');
```

```
    }
}
```

But, in this context, it will be an anti-pattern Service Locator. Depending on a container hides all the information about class dependencies. If we want to test this class, we should examine it and find out all the dependencies that it resolves out of the container and mock them for every test. We also cannot be sure that all the dependencies are already correctly registered in the container. In this way, we lose all advantages of Dependency Injection. Objects shouldn't know about the container.

A better option is to break `OrderController` into several small classes, each with its own responsibility, like `PaymentController` or `ShippingController`.

Cyclic Dependencies

Class A depends on class B, but class B depends on A.

```
class A
{
    private $b;

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

class B
{
    private $a;

    public function setA(A $a)
    {
        $this->a = $a;
    }
}
```

This sort of dependency often means that you don't really have two independent classes. Chances high, that these classes share one common responsibility, so maybe they both should be one class, rather than two separate.

Injecting Dependencies For Other Objects

An object receives a dependency that it doesn't actually use, instead it simply passes this dependency to another object.

```
class DeliveryController
{
    private $http;

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

    public function sendOrder($orderId)
    {
        $order = Order::find($orderId);

        $delivery = new Delivery($this->http);

        $delivery->sendOrder($order);
    }
}
```

`DeliveryController` depends on `HttpClient` and doesn't actually use it. The only reason to depend on `HttpClient` is to pass it later build an instance of `Delivery` class. The problem here is that we are coupled to `Delivery` class. It is hardcoded here with the `new` keyword. But even if we are not going to use another class here, we are still coupled to `Delivery` constructor. If its signature changes, the code here will be broken. We perfectly know this scenario: *"I fixed it here, but now it is broken there"*. To fix this issue, we can inject an already constructed instance of the `Delivery` class.

```
class DeliveryController
{
    private $delivery;

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

    public function sendOrder($orderId)
```

```
    {
        $order = Order::find($orderId);

        $this->delivery->sendOrder($order);
    }
}
```

In other cases when you need to create a new object and pass some dependencies for it, factories can be a good solution. Simply inject a factory, that already has all the required dependencies. And then use it to build an object. Next time, when the process of instantiation changes, you should change only one line of code in the factory, instead of using *find in project* in your IDE.

Maybe Your Don't Need It

Always think first before making any serious design decisions. Don't follow any rules blindly. When all these *gurus* on Twitter post articles about DI and discuss it's advantages, but you have some hardcoded dependencies in your code, that doesn't automatically mean that you are not a professional developer. As any other technique, it has pros and cons. Previously we have discussed dependency injection only in a positive light. However, you can add all these layers of indirection for dependency injection to solve the problem that doesn't really exist. If your class uses a dependency, but that concrete implementation is the only one available in your system, then it will be more painful to create a loosely coupled solution, rather than just use *new* keyword and forget about it.

Maybe you have `Api` class and it uses Guzzle client to send HTTP requests. We can simply hardcode this client in the constructor and use it.

```
namespace App\Api;

use GuzzleHttp\Client;

class Api
{
    private $client;

    public function __construct()
    {
        $this->client = new Client();
    }
}
```

But let's see what happens if we follow *good practices* and use dependency injection in this case. At first, we need an interface `HttpClient`. Then we need to wrap a Guzzle client instance into a wrapper (adapter pattern), which implements this interface. Then we need a factory or dependency container, which will create this wrapper for us. And at least we can pass it to the constructor of `Api` class.

```php
interface HttpClient
{
    // ...
}

class GuzzleAdapter implements HttpClient
{
    // ...
}

class Api
{
    protected $client;

    public function __construct(HttpClient $client)
    {
        $this->client = $client;
    }
}
```

And now ask yourself one question: *what is the probability that you will use another HTTP client implementation, instead of the Guzzle client?* In six months you will change it to Curl? I don't think so. Is it worth it? You have created all these classes and complexity to replace only one line of code, which is unlikely to be changed.

Conclusion

You can think about dependency injection like the classic avoid of using hardcoded constants in your code. For example, when you have a database name you often move it to a config file, rather than using it as a hardcoded constant. Then you pass it as a variable which stores that value to the code where it is needed. We do it because a database usually changes more frequently than the rest of our code (production database and a testing one). So, the same is true in the object-oriented world. In OOP instead of hardcoded constants we have *objects*, but the reason, why we move the creation logic from the class stays the same - *the objects change more frequently then the code that uses them* (and again we can use tests example here, where we want to use some different objects).