Practical Design Patterns In PHP

By Brandon Savage

Copyright

This version of the book was published on March 31, 2014.

Credits

This book was edited by Diana Ecker. Technical editing was provided by John Mertic and Chris Tankersley. Thanks to all three of them for their dedicated and wonderful work in producing this book and making it the best it could be.

The iterator concept was adapted from a video produced by Anthony Ferrara. You can watch the whole video on YouTube at http://www.youtube.com/watch?v=tW6GcZjBc3E

Thanks to my loving wife Debbie, for putting up with me as I worked through this project for several months. It means the world that she believes in me.

About the Author

Brandon Savage is a software developer who began his PHP career in 2003, before PHP 5 was even released. He began by developing gaming systems, figuring that having a computer execute complex math was preferable to doing it by hand.

Brandon eventually figured out that he could make money by writing code, and began full time software development in 2008. He quickly outgrew his first few roles, and eventually landed at Mozilla, where he works as a Software Engineer on the Socorro team (Socorro is Portuguese for “help”, and Socorro is the software tool that aggregates Firefox crash data).

Brandon lives in Olney, Maryland with his wife and three cats. He is a private pilot and loves aviation. You might see him flying his red-and-white Piper Cherokee 235 over the skies of the East Coast. He promises not to fly too low, though.

Find bugs?

Every effort was made to make this book as accurate as possible. But it’s possible that bugs, typos and other errata may still exist. If you find something, please file a bug!

https://github.com/tailwindsllc/practicaldesignpatternsinphp

TABLE OF CONTENTS

CREATING SIMPLE OBJECTS

Chapter 5

CREATING SIMPLE OBJECTS

The process of object creation is a difficult subject for many developers. In fact, creating objects is considered a responsibility in and of itself. So how do we actually go about creating objects that we're going to use in our application?

The solution is to abstract out the object creation process, delegating it to an object whose sole responsibility is to create those objects we depend on. And for the creation of relatively simple objects, there are a few design patterns we can rely on, including the Abstract Factory, Factory Method and the Singleton.

These patterns are specifically designed to create relatively simple objects, but to delegate the actual creation of an object to the class
It's an age old question many developers struggle with all the time: if I am working to invert my dependencies and not create objects inside my classes, how do I go about creating the dependencies that I need during runtime that can't necessarily be injected?

Developers struggle with this issue all the time. They recognize the need to use dependency inversion and interfaces, so they also know they shouldn’t create objects inside other objects. But they also know that objects have to come from somewhere, and struggle with the need to create objects at runtime. Figuring out how to create the objects they need without injecting them is a frustrating problem.

The answer to this dilemma lies with several design patterns used for creating simple, straightforward objects. The principle is simple: delegate the creation to another object and allow that object to decide what to create and how to create it. The object is then created on demand, but the requestor is not responsible for the creation; another object or method is.

There are three patterns used for creating relatively simple, straightforward objects: the Abstract Factory, Factory Methods, and the Singleton Pattern.

The Factory Method

A factory method is simply a method in an object that is responsible for creating and returning another object. Many developers have seen and used factory methods before; they can be very simple and straightforward, or complex and lengthy. The purpose of The Factory Method is to create objects and abstract that creation process from the requestor.

A factory method is usually part of a design pattern called the Abstract Factory, but it doesn't have to be. Other design patterns, like the Singleton, use The Factory Method Pattern as well. We'll be taking a closer look at these patterns shortly.

A factory is relatively simple to write. All that's required is some method that creates and returns an object, and can be as simple as this:

```php
<?php

public function factoryMethod() {
    return new Object();
}
```

It's a relatively simple, straightforward process to create a factory method. Our simple example here illustrates that a factory doesn't have to be complex, even though many factory methods are more complicated than this. Most factory methods are part of a larger object or part of a larger set of design patterns.
Many older frameworks (and some newer frameworks) use static methods for factory methods, creating something that looks like this:

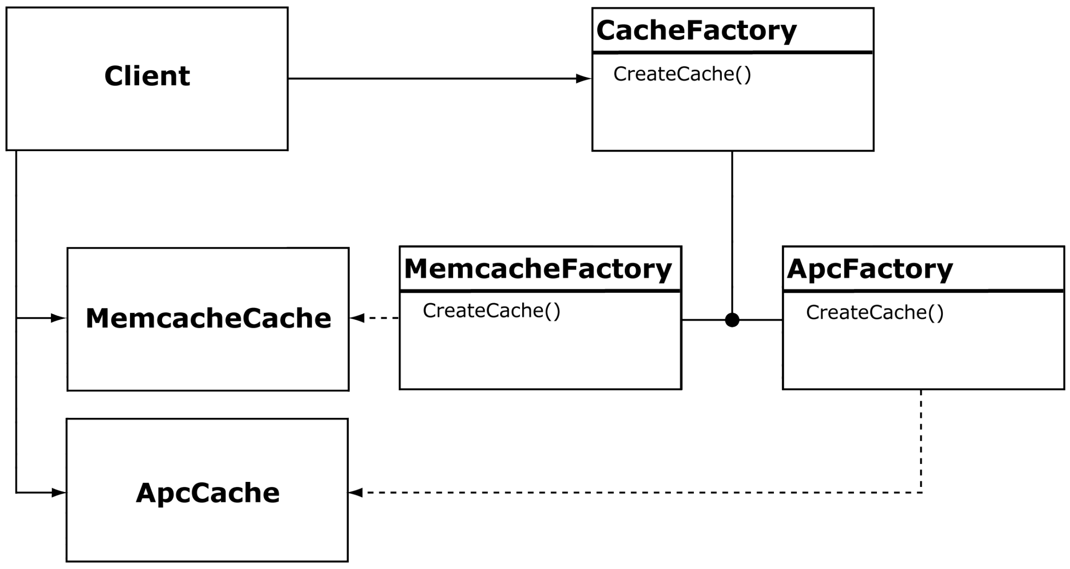
```php
<?php

public static function factoryMethod() {
    return new Object();
}
```

The method is accessed with the ObjectName::factoryMethod() call, but this methodology is largely frowned upon in more modern frameworks. First, by tying the class name to The Factory Method you've essentially created the tight coupling you were hoping to avoid with The Factory Method. Also, it's difficult (if not impossible) to properly mock The Factory Method and create a mock object that can be used in unit testing. Though some frameworks make use of static methods, static methods are something best avoided if at all possible.

The Abstract Factory Pattern

What happens when you have a collection of similar objects that you want to be able to instantiate, but you want to be able to swap them out at runtime? For example, imagine that you want to create a caching system, but you want to support multiple engines for the caching system. How do you go about doing this?

The answer lies with the Abstract Factory Pattern. The Abstract Factory Pattern allows an object (called the Client) to be given the factory it will use, without having to care about which concrete factory it has been given. Instead, it only cares about two interfaces: the interface of the Abstract Factory and the interface of the Product. Our caching example might look something like this:



In this example, the CacheFactory is extended by the concrete factories of MemcacheFactory and ApcFactory. The idea is that the Client knows the CacheFactory interface, but it's totally unaware of the inter workings of the concrete factory objects. The Client expects that it will receive an instance of Cache, and knows that interface, but does not care about the internal details of the specific concrete cache it's working with. In this way, the Client is completely abstracted from the creation process.

This is exactly what we want: we want the Client to be clueless about how the objects are created, and to delegate the creation of the objects to the specific factories that create them. Once the objects are created, they are sent back to the Client. The Client knows nothing about how the cache works, and only cares that the interface is consistent and expected.

Of course, this can be overkill for creating really simple objects, which is why The Factory Method exists at all. Also, it is totally inadequate for creating large, complex objects that require multiple steps or configurations, which makes it unsuitable for these kinds of situations as well.

Sample Code for the Abstract Factory

Let's use our cache example as our sample code here. Which cache we use is often defined by configuration, and the creation of the cache object is handled at runtime. The Client needs to know two things: the interface for the abstract factory (Modus\Cache\Interfaces\CacheFactory), and the interface for the product (Modus\Cache\Interfaces\Cache). While PHP allows us to type hint for the abstract factory, we cannot type hint on the return value of an object; thus, we'll need to assume and rely upon the interface in a less concrete way.

```php
<?php

namespace Modus\Clients;

use Modus\Cache\Interfaces;

class Client {

    public function __construct(
        Interfaces\CacheFactory $cacheFactory
    ) {
        $this->cacheFactory = $cacheFactory;
    }
    public function getCache()
    {
        return $this->cacheFactory-
>getCache();
    }
}
```

Our Client now has all the things it needs to utilize the abstract cache factory. Let's define the abstract cache factory and the cache interface.

```
<?php

namespace Modus\Cache\Interfaces;

interface CacheFactory {
    public function getCache();
}
```

That takes care of the factory; now for the cache itself:

```
<?php

namespace Modus\Cache\Interfaces;

interface Cache {

public function set($key, $value, $ttl =
3600);
public function get($key);
public function delete($key);
public function purge();

}
```

Now we have the interfaces that our Client will rely upon. Let's now go ahead and create some concrete instances of our Factory:

```
<?php

namespace Modus\Cache;

Use Modus\Cache\Interfaces;

class MemcacheFactory
     implements Interfaces\CacheFactory {

     public function __construct() {
          // TO DO: CONFIGURATION!
     }

     public function getCache() {
          return new MemcacheCache();
     }
}
```

We can just as easily create an APC cache by changing the code slightly:

```
<?php

namespace Modus\Cache;

Use Modus\Cache\Interfaces;

class ApcCacheFactory
    implements Interfaces\CacheFactory {

    public function getCache() {
        return new ApcCache();
    }
}
```

Note that the MemcacheFactory has a configuration requirement that the APC cache does not; this is okay! Since the factories are created and injected into the Client at run time, we can handle the fact that the factories might have slightly different configuration requirements elsewhere. Additionally, the Client doesn't have any need to know about the configuration requirements, which further encapsulates the abstraction of the Cache from the Client.

The specific implementation of each product is unimportant to the pattern; we will revisit the cache example when we discuss adapters. However, because the Client depends upon the interface for the product, it is imperative that we honor that interface (see the Open/Closed Principle and the Liskov Substitution Principle).

The Singleton Pattern

The Singleton Pattern is perhaps the most well known - and most often misused - pattern in all of PHP design pattern development. Its simplicity, combined with its seeming benefits makes it a widely-used (and overused) pattern. The Singleton is not so much a recommended pattern, as a pattern I recommend you shy away from. But it's important to understand how the Singleton works, and what its drawbacks are, so that you can make an informed decision.

The Singleton is an object that allows only one instance of itself during runtime, and proffers a global access point to itself. In PHP, this means that the Singleton sets the __construct() and __clone() methods to private, and the class to final. In order to provide a global access point, the Singleton implements a static factory method. Expressed in code, the Singleton looks something like this:

```
<?php

class MySingleton {

    private static $instance;

    private function __construct() {}

    private function __clone() {}

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

}
```

There are few design patterns that always look a certain way; in fact, most can be implemented in a thousand different ways to reflect a thousand different use cases. But the Singleton Pattern is unique because there is truly only one way to represent it in PHP (at least, if the developer is doing it correctly).

The Singleton comes with a host of problems. The Singleton is impossible to extend: the private methods and final keyword ensure that PHP won't let us. Also, the global nature of the Singleton means it's difficult to test, because the global changes made to the Singleton are reflected across all tests, not just the area under test. This can result in test failures, even when the test that fails is not the problem.

Should the Singleton be marked "final"?

The goal of the Singleton Pattern is to ensure that only one Singleton exists at any given time. Yet PHP allows for developers to redeclare the __construct() and __clone() methods as public in subclasses, thus potentially defeating the intent. If a developer marked the class or methods as final, this wouldn't be possible. Should the Singleton class be marked as final?

The answer is no. The Singleton Pattern is about implementation, and the specific quirks of the language, while taken into account, aren't sufficient for forcing the Singleton to be marked final. There are many legitimate reasons for extending a Singleton (like adding functionality) that wouldn't be possible if the Singleton is marked final.

But the Singleton’s most damning problem is the fact that it violates just about every SOLID principle. The Singleton creates itself, meaning that you can’t abstract away the creation responsibility (violating the Single Responsibility Principle). It can’t be replaced with a subclass of itself (Liskov Substitution Principle), and the global availability violates Dependency Inversion too. The Singleton can’t be extended (Open/Closed Principle), and the lack of interface makes it impossible to follow the Interface Segregation Principle. The Singleton Pattern has its uses, especially in desktop software (the Gang of Four talk about printer spools as a use case for the Singleton), but in web development, the Singleton pattern has few if any valuable uses.

What are those “valuable uses?” Like I said, there are a few. For example, the Registry Pattern (not covered in this book) often makes use of the Singleton for ensuring that no more than one registry exists at runtime. Also, you may choose to use a Singleton for a preferences or configuration object. The configuration object is unlikely to change state, eliminating many of the concerns associated with the Singleton in testing. However, these limited uses are often not sufficient to overcome the Singleton’s problems, and thus the Singleton is often best avoided altogether.

What about more complex tasks?

We have largely focused on the creation of simple, straightforward objects that require little, if any, configuration and can be instantiated in a single step. This isn’t always the case. In the next chapter we’ll examine some of the methods for creating complex objects that may require more than a single step, or have difficult configuration requirements.