# Practical Design Patterns In PHP

By Brandon Savage

## Copyright



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

Books like this are possible by the time and investment made by the authors. If you received this book but did not purchase it, please consider making future books possible by buying a copy at http://www.practicaldesignpatternsinphp.com/ today.

## 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.

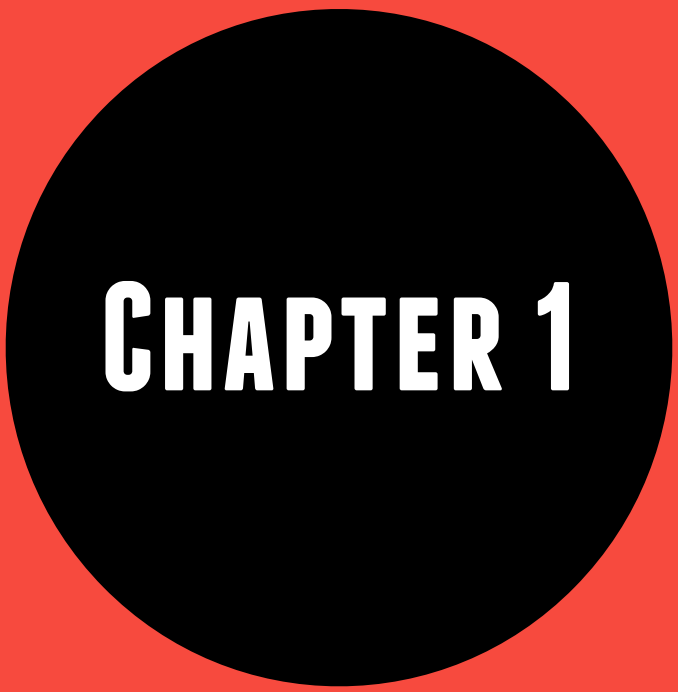
## Find bugs?

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

# THE NEED FOR DESIGN PATTERNS



# THE NEED FOR DESIGN PATTERNS

Writing object-oriented code is hard. Writing good, reusable, well-designed object-oriented code is even harder. In fact, the thought process for writing object-oriented applications feels foreign, almost wrong. It seems to be at odds with what we've learned throughout our entire careers thus far.

We use indirection instead of bundling everything together. We abstract and delegate rather than having one big function do all the work. We obfuscate complex idea, passing them to different layers, rather than trying to bring clarity in the layer that needs that bit of code. We take code that would generally all be shoved together and fling it into outlying objects and helper classes, while the meat of our application ties it all together.

Why do we do this? Because we want to create objects. But more than that, we want to create reusable objects - objects that can be taken out of the context they were originally designed for and used in a new one. Object-oriented development empowers us to create libraries of code that can

be used over and over again, to swap components around, and to use configuration instead of code changes to manage our applications.

Yet for a relatively novice developer coming into object-oriented design for the first time, this can be quite overwhelming. Many developers, especially in PHP, assume that by wrapping their procedural code in classes, they have somehow bridged the gap between procedural programming and object-oriented design. But, in fact, they have not. It takes time to grasp the principles and processes of object-oriented design. Wrapping procedural code in objects isn't object-oriented design at all.

So how can a novice or intermediate developer move forward in object-oriented design? We don't want to start from first principles each and every time. Experts don't do that, either. In fact, over time experts have recognized some best practices and patterns that they can use repeatedly to create compelling, cohesive, exceptional object-oriented code. And it is these practices and patterns that novice developers can learn.

Experienced developers aren't starting from scratch every single time they write an application. We know some things that work, and we have a history with things that don't. We bring our experience to the table each time we start with a

blank screen. And we use that experience to get better and faster.

Experts have reached a point where they come to the table with tremendous amounts of knowledge as to what works and what doesn't. They've made mistakes in the past, and know how to avoid making the same mistakes in the future. They are focused on the task at hand, and design decisions seem to flow from their minds automatically. This is the hallmark of their experience: they've been there before. They've seen this already.

Becoming an expert takes time and effort. There's no shortcut to reaching that level. But even if you are not an expert, you can still have access to the patterns and practices that experts have come to know. This is what design patterns are all about: codifying and presenting these "best practices" in an easy-to-use, accessible way.

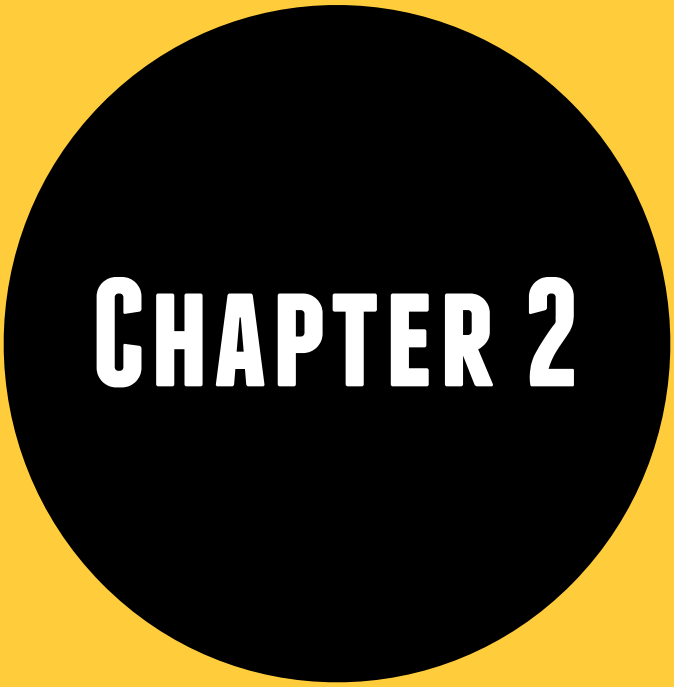
Could you learn these things on your own? Sure. Many experts have; they've learned by doing, and by making painful mistakes in the past. The problem is that to get there, you have to take the long, hard route. But this book is designed to be your shortcut.

Instead of expecting you to make each of the mistakes yourself or come up with each of the patterns on your own, this book

provides an opportunity for you to cut to the front of the line and be equipped to understand patterns and design applications from a new perspective. The goal is to level up your object-oriented abilities, and the focus is on showing you the design patterns you need in a language that's familiar to you.

As you work through the material, you'll find that, more and more, the principles and practices of object-oriented applications become familiar, almost second nature. My goal is to help you connect ideas together in a way that makes sense and offers you new options for your own development practices. This book is designed and organized to take the basic object-oriented practices you already know, expand upon them, and then demonstrate in concrete, actionable ways exactly how you can move forward in your own development practices. Rather than making costly mistakes on your own, you have an opportunity to learn from the mistakes of others and to become a better developer. I'm excited that this book will be part of that journey for you!

# WHY ARE DESIGN PATTERNS SO #?*! DIFFICULT?



# Why Are Design Patterns So #?*! Difficult?

“Let’s talk about design patterns.”

That sentence alone is enough to strike fear into the hearts of developers everywhere. Most of us hate design patterns. Well, hate is a strong word. We avoid design patterns. They’re hard. They’re abstract. They’re difficult to understand and feel impossible to implement. The only person that seems to love design patterns is the guy who drones on about them at our annual conference. Sure, we go to that talk. We use it to catch up on our sleep or get over the hangover from the night before.

The cold, hard truth for most of us is that design patterns seem to be the one concept that we can’t fully grasp. We know they’re important. We know they are useful. But the resources out there make design patterns almost impossible for us to understand. And that is really frustrating. So we avoid them.

But I think it’s time that you took another look at design patterns.

You hate design patterns not because design patterns are bad, but because they're misunderstood. You likely see design patterns as abstract concepts that have no practical bearing on your life. Some mythical creature to be avoided, or at least talked about with reverence and awe. But this simply isn't true.

In fact, you have almost certainly implemented design patterns before without even knowing it. How do I know this? Because a design pattern is nothing more than a common way to solve a common problem that is shared by developers everywhere.

Hear the good news: *somebody has already solved your problem!*

That's right: your problem has already been solved by somebody else somewhere. Many of your problems have already been solved by another developer somewhere else doing similar work. You don't have to bang your head against the desk or start from first principles anymore. You can enter the Promised Land!

Let me back up and explain.

Design patterns are less about having concrete, finished code solutions, and more about having great concepts and designs for solving common problems. Need to make the different interfaces of two objects the same? There's a pattern for that.

Need to handle a specific request in accordance with a set of rules? There's a pattern for that! Want to simplify a subsystem and create a simple interface? There's a pattern for that, too.

Sure, you still have to write the code. But you don't have to reinvent the wheel. You don't have to create the architecture from scratch. You don't have to agonize over exactly what objects you're going to create or how those objects will fit together. The hard work has been done for you. Now you get to take that pattern and implement it. No more agony and pain. You get to focus on your problem, not your architecture!

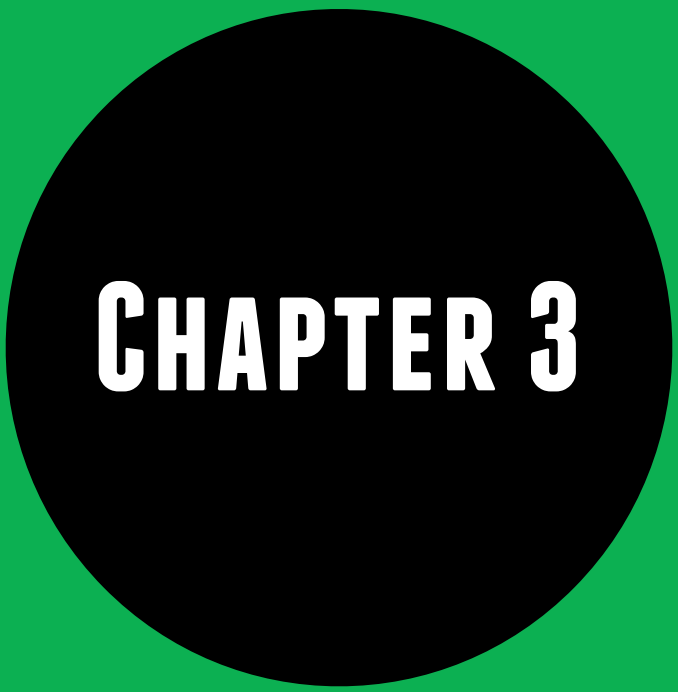
Design patterns are about saving time, saving effort and saving energy. Experts use design patterns because they have learned that certain patterns work very well; they know that creating a custom solution will take longer and end up looking like an established pattern anyway. Experts see this as a waste of time; they've taken the time to learn the patterns that work for them, and use those patterns over and over again to improve their code. They rarely start from first principles, because they have no need to. Design patterns are their solution.

Hear the good news:
*somebody has already solved your problem!*

That's what design patterns can offer you. And this book is designed to show you how to use design patterns in your own development process, so that you can design code faster, develop code more accurately, and even head home on time every night.

It's easy to look at design patterns and think, "They are too hard; I'll never get them." But you can "get them," and this book will show you how. So come along, and learn about the design patterns that make object-oriented development easier, better and far more organized than anything you come up with on your own.

# WHAT ARE DESIGN PATTERNS?



# WHAT ARE DESIGN PATTERNS?

The phrase “design patterns” carries many meanings, and its definition changes based on which group of people you’re talking to at any given time. Different developers see design patterns differently, even if they are using the same language. In order to effectively describe and use design patterns, it’s important to define what a design pattern is, as well as what a design pattern is not.

For our purposes we are going to assume that design patterns have five distinct components that we can utilize in helping us to understand them more fully.

## Design patterns solve common problems

When we talk about design patterns, we’re talking about common ways that we can solve problems. After all, this is exactly what a design pattern is: an agreed-upon way to solve a common programming issue.

Expert developers know that they can’t go back to the drawing board each and every time they want to architect an

application. Instead, they recognize that it is vitally important for them to use patterns and practices they have already learned and found to be useful. In fact, one of the biggest differences between a novice programmer and an experienced one is how they approach a problem. A novice programmer may start from first principles, but the expert knows certain patterns that already make sense, and has a toolkit readily available to help solve the problem at hand. This difference in approach can save hours for an expert.

It's important to recognize that design patterns aren't designed for one-off solutions. In fact, for one-off problems, there may not be a design pattern that fits. But the beauty of design patterns isn't their universal application; it's that for the problems they solve, they make implementing a design and a solution that much simpler, allowing developers to focus on the truly hard problems they face.

Additionally, the word "common" is at the crux of understanding the definition of a design pattern. These aren't problems that I came up with or that you'll face alone. They are problems that developers the world over have identified, wrestled with, and solved in a relatively uniform manner. Design patterns are about common solutions to common problems.

The beauty here is that someone else has already solved your problem. The hard work has been done! Instead of focusing on and fighting with architecture, you can instead focus on solving the interesting parts of your problem, knowing that the patterns are there to back you up.

## Design patterns can be used in many different situations

Since design patterns are specifically intended for solving common problems, there are lots of different ways to implement each one. In fact, it's possible to implement the same design pattern hundreds of times but never to implement it quite the same way twice. This speaks to the unique nature of design patterns: they are concepts, not blueprints; ideas, not finished designs.

To illustrate, take this analogy. Engineers and architects are called in regularly to design new buildings and develop land. Even though the principles they follow are nearly the same in all cases, the specifics are different. Land size differs. Intended use case is unique. Even where developers use the same patterns over and over again, the specific implementation of a particular plan is as varied and unique as the number of projects an engineer faces.

That's not to say that engineers and architects don't follow patterns. They do! Engineers follow the same patterns over

and over again to solve common problems. The principles are set, established, defined. Their implementation, though, is unique to the project at hand, just as your implementation of a design pattern will be unique to the problem you are solving.

In software, the patterns are different, but the idea is the same. You implement patterns as our established, defined principles, and focus not on the structure but on the specific issue we have to solve. This greatly simplifies and dramatically reduces the effort and time we spend on particular issues.

## Design patterns are best practices

In fact, it's possible to say that design patterns are established best practices that we can use over and over again to create code and projects that are easily understood by others. Developers creating or observing code that implements a particular design pattern will have a specific understanding of how that code is supposed to work, even if the specific details are uncertain. This is the power of design patterns: they add clarity to an otherwise difficult situation.

These best practices are best practices because they have been agreed upon over time by thousands of developers. There's no need to reinvent the wheel; the wheel has been invented, but left flexible enough to adapt to almost any situation one can devise. Just as there are millions of unique, individual buildings worldwide, there can be millions of unique,

individual implementations of, say, the Strategy Pattern or the Abstract Factory.

Even though a particular design pattern is a best practice doesn't have to mean it's an absolute practice. There's room for variations, since every problem is different. Just as it's possible to implement a single pattern in thousands of different ways, there are ways to bend patterns to fit your particular challenge, or to work with the pattern in a way that maybe nobody else has before. Once you understand best practices, you also learn how you can bend them to suit your needs!

## Design patterns are not finished code

Even though design patterns represent best practices, they don't represent specific code implementations. In fact, many design patterns can't be expressed in code at all. This creates a bit of a conundrum for developers, since they are used to seeing concepts expressed in code. When a design pattern isn't expressed in code, that's extremely frustrating.

The reason that design patterns cannot be adequately expressed in code is that they are unique to the situation in which they are implemented. They are not application- nor implementation-specific in the same way that actual computer code is. Presenting a design pattern in code alone would result in an impression that might not be completely accurate.

Of course, to write a technical book without illustrating some of the ways a design pattern might be represented in code wouldn't work very well. This book contains code samples, but the code samples here are for illustration purposes only. They are in no way the only implementation of a particular design pattern; in fact, they may not even be the best implementation of a particular design pattern. Instead, the code samples here are designed to illustrate a particular way to implement a pattern, and to show you what you can expect when you do it for yourself.

You'll find that as you go forward and implement patterns, sometimes they will look similar to the code here, and sometimes they'll look completely different. That's okay! Design patterns are mostly about how objects interact with each other, and are less focused in the specific code that accomplishes that interaction. If design patterns were code-based, then certain design patterns would be language specific, and that would make for a lousy model of reuse.

## Design patterns are not mutually exclusive of one another

The final distinguishing feature of design patterns is that they are not mutually exclusive of one another; that is to say, implementing one design pattern doesn't eliminate the

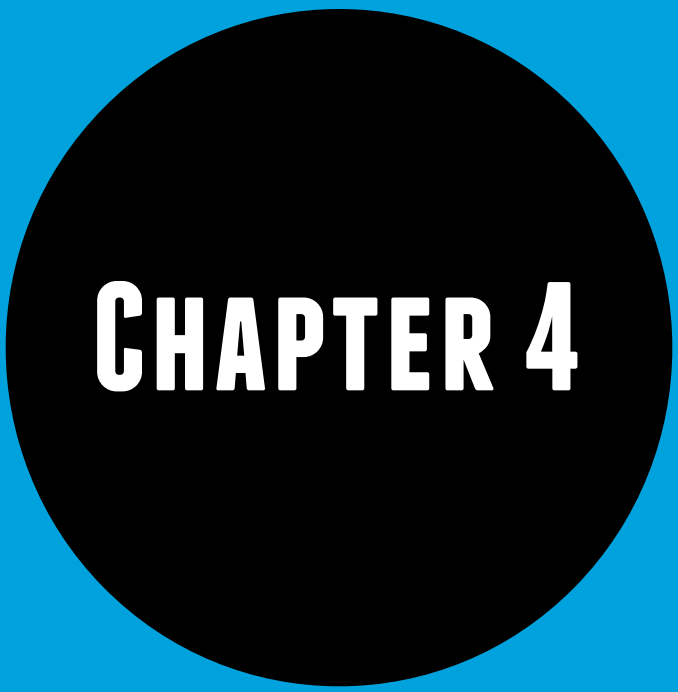
possibility of implementing another pattern alongside it for the purposes of creating a hybrid pattern.

Many patterns rely upon one another to create cohesive, comprehensive structures in applications. Design patterns don't exist in a vacuum, and they don't exist independent from one another. Using one pattern doesn't preclude use of another, and you should feel free to combine as many patterns as you like to achieve your desired results.

In fact, many times the most powerful components of code combine two or even three design patterns. For example, it's possible to take the Publish-Subscribe Pattern and mix it with the Chain of Responsibility Pattern to create really great logging software, or the Facade Pattern with the Mediator Pattern to create interesting collections of objects at low levels of our application. There's no reason to think that we can't take one pattern and apply it alongside another pattern, because this is encouraged!

Even though design patterns may not fit a specific aspect of code or fit into our preconceived notions, they are useful, powerful concepts that will improve your understanding of object-oriented development and design. But before we dive into design patterns, let's take a look at the object-oriented design principles we're trying to preserve through the use of these patterns.

# THE PRINCIPLES OF OBJECT ORIENTED DESIGN



# THE PRINCIPLES OF OBJECT ORIENTED DESIGN

In order to understand design patterns, you must first understand the principles behind object-oriented design. While there are many ways of expressing these principles, in this book we'll focus on the five principles expressed in the acronym **S.O.L.I.D.**

The SOLID principles are a group of concepts that have been established as best practices and represent a way of creating and developing objects that exhibit desirable behaviors like loose coupling, single responsibility and ease of reuse.

The SOLID acronym stands for:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Let's examine each of these principles in more depth.

## The Single Responsibility Principle

Most developers are familiar with this principle; in fact, you've probably heard it mentioned at just about every object-oriented talk you've ever attended. But many developers still struggle with understanding this principle, so it's important to review.

The Single Responsibility Principle states that each object should have one responsibility, and that the given responsibility should be fully encapsulated by the object.

What this means, in practice, is that object-oriented developers must identify the responsibility of a specific object, and then work to ensure that objects only handle that responsibility. Additional responsibilities should be delegated to other objects.

Inevitably, this definition brings up a question: "What is a responsibility?" A responsibility is a clearly defined aspect of functionality that can stand alone and be reused within code. For example, an algorithm, a database connection or a cache object can all be considered a single responsibility. Additionally, object creation and generation is a responsibility in and of itself, and there are several creational patterns that we'll examine in this book.

Enforcing a single responsibility rule on objects encourages reusability, discourages "God objects" (large objects that do

too much) and prevents us from turning our procedural code into procedural code wrapped with class statements.

## The Open/Closed Principle

When developing APIs and objects, it's important to know what is available for modification and what is not. The Open/Closed Principle helps establish this by expecting that classes are open for extension, but closed for modification.

Over time, this has been taken to mean one of two things. Some developers follow a practice known as Meyer's Open/Closed Principle. This principle states that once the implementation of an object is complete, the class can only be modified to correct errors and cannot be modified to add new features. Such modification requires that the class be extended (the "open for extension" component). The other interpretation of this principle is the Polymorphic Open/Closed Principle. This principle states that once solidified, the API of a class is set, but classes inheriting from the parent may implement that API in any way they so choose. This is a much more popular interpretation of the Open/Closed Principle.

For the purposes of this book, we will mostly focus on the Polymorphic Open/Closed Principle. Many of the design patterns we will discuss rely on inheritance or interface implementation to accomplish their goals.

## The Liskov Substitution Principle

This principle was developed by Barbara Liskov and was originally introduced in 1987 in a conference keynote. The principle, in effect, states that one object should be replaceable with a subtype of itself without altering the correctness of the program.

To put that a different way, let's say a developer has a class of A, along with two objects which extend A: B and C. The developer should be able to substitute B for C without needing to modify the program.

Why is this principle so important? Because it forms the basis for the creation of common interfaces, which is a crucial aspect of the development of design patterns. Many patterns rely on one object understanding and knowing the interface of another to accomplish certain tasks.

It's important to note that the Liskov Substitution Principle prohibits one object from understanding the internals of its dependencies, since knowing the internals of the dependency would inevitably result in the objects being tightly coupled, and would keep object substitution from being possible.

As such, the Liskov Substitution Principle is the glue that ties the SOLID principles together (fitting, since L is the middle letter in SOLID).

## The Interface Segregation Principle

"God objects." We've all seen them. They're the objects that do everything and then some, that have a hundred public methods and too many properties to count. These methods are huge and unwieldy, with an API to match.

These objects are far too large. They probably violate other aspects of the SOLID principles, and they most definitely violate this one, too.

The Interface Segregation Principle states that objects should only rely on the methods they actually implement, and shouldn't be forced to implement methods they don't need just to satisfy an interface.

## In other words, smaller interfaces are better.

This has some serious practical implications. For example, developers of a database API might think twice before making transactions part of their base interface, because not all databases or database types supports transactions. Instead, creating a TransactionAware interface and a Database interface (and then combining the two when necessary) would be more appropriate.

While this aspect of SOLID usually plays a minor role, it's an important principle to understand and will pop up from time to time in the design patterns we're discussing.

## The Dependency Inversion Principle

Many PHP developers mistake this principle for the Dependency *Injection* Principle, but dependency injection is only one piece of the puzzle here.

The Dependency Inversion Principle states that objects should depend upon abstractions rather than concrete instances of their dependencies. In other words, objects should understand the *interface* of a dependency, but not care so much about the *implementation* of a dependency.

In many cases, developers assume that dependency injection satisfies this principle. The problem with this belief is that dependency injection only satisfies part of the principle. By injecting the dependencies we are in fact inverting control. And by type hinting we are ensuring that the dependencies' type (interface) is checked.

But PHP makes an assumption that's inherently flawed when it comes to defining an object type: it assumes that the type is the same as the class name. Thus, if ClassA implements InterfaceA, type hinting for ClassA or InterfaceA will both yield

valid results, but only one of them properly follows the Dependency Inversion Principle.

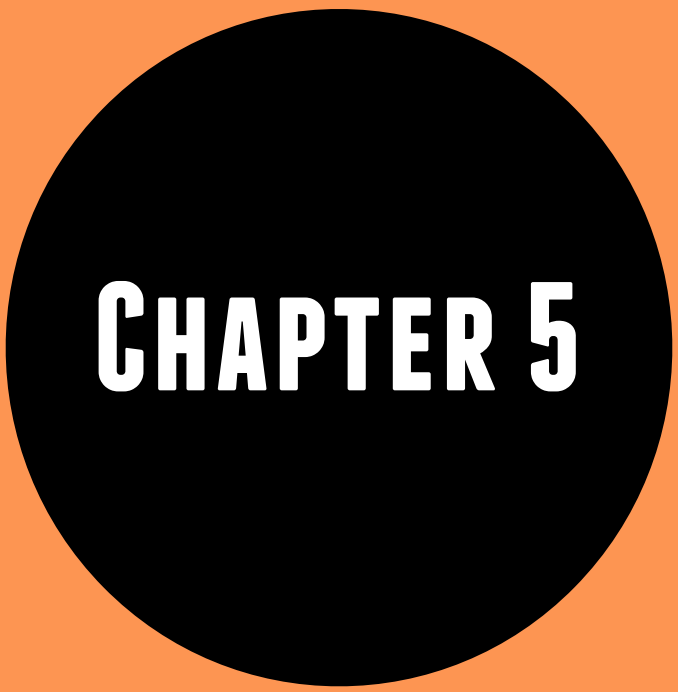
When type hinting, especially in PHP, we want to carefully consider whether or not we are type hinting on a particular concrete object (class name) or if we're type hinting on an interface or abstract class (abstractions). The former is incorrect, while the latter is appropriate - and its the goal of the Dependency Inversion Principle.

Many of the design patterns we're working with here are designed in such a way as to fully encapsulate and encourage dependency inversion throughout the entire stack. The benefits will become obvious as we see how objects can quickly be changed for one another at runtime or in development.

## Tying it all together

Some design patterns more closely adhere to the SOLID principles than others. As we discuss the design patterns, we will also examine how the design pattern follows (or ignores) the principles of SOLID. Some patterns follow the principles 100% of the time while others don't. It's important to understand how each design pattern works, and to understand how the pattern encourages (or discourages) good design in order to make independent decisions about how and when to use them.

# CREATING SIMPLE OBJECTS



# 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

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:

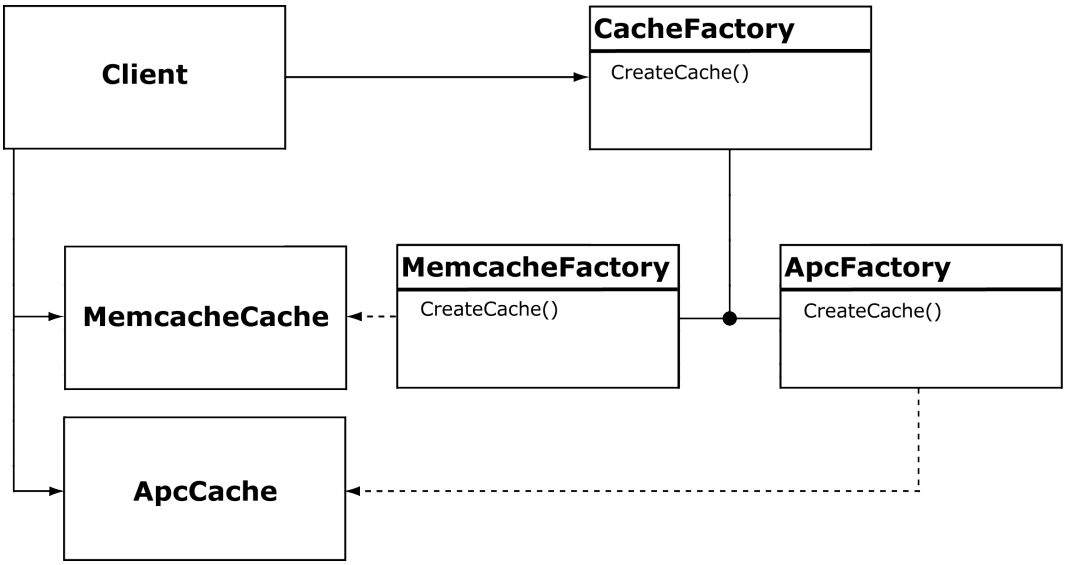
```
<?php

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

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.

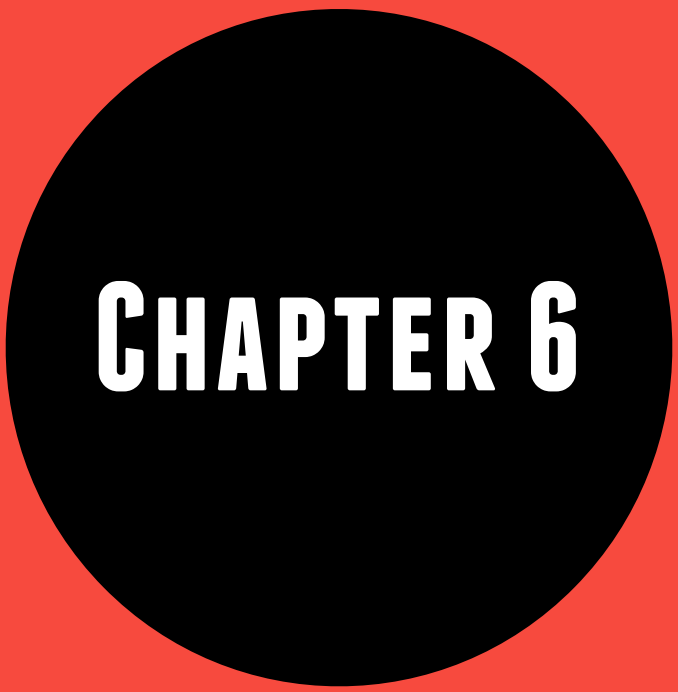
But the Singleton's most damning problem is the fact that it violates just about every SOLID principle. The Singleton creates itself, meaning that you can't abstract away the creation responsibility (violating the Single Responsibility Principle). It can't be replaced with a subclass of itself (Liskov 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.

# CREATING COMPLEX OBJECTS



# CREATING COMPLEX OBJECTS

The Abstract Factory pattern is a great pattern for creating simple, straightforward objects for which the configuration and contents can be abstracted away to other parts of the application. Unfortunately, not every object is straightforward or simple. Sometimes objects require a tremendous amount of interaction with the Client object during the creation process. These objects might need information or configuration that the Client has or will have, but that is not conveniently available when the Client invokes the creation process. Or they might require data that can only be generated through a series of operations that can't be executed until the object creation process has started, creating a blocking process. In these cases, we want to use a different pattern.

The way we solve this problem is called the Builder Pattern. This pattern allows us to build objects step by step, rather than all at once. A chief difference between the Abstract Factory Pattern and the Builder Pattern is the fact that the Builder returns the object to the Client only when the Client asks for it,

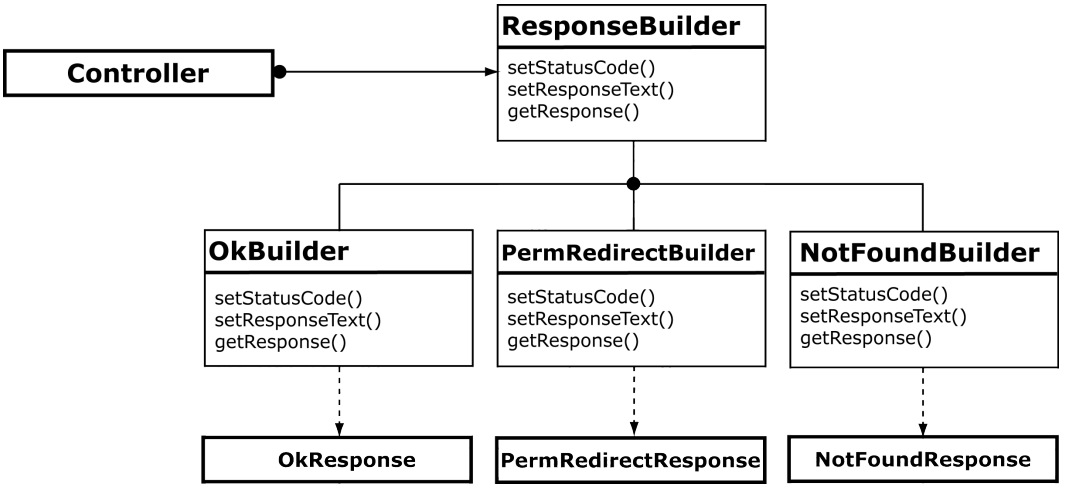
allowing us to complete the process of construction in phases, rather than all at once.

## One of these objects is not like the others

Some objects are particularly challenging to build. For example, an HTTP response may have several parts to it: adding headers to status codes, creating page templates, and HTML layout. Still, it would be useful to abstract the creation process for an HTTP response to another object, which can be responsible for the implementation and execution of the process.

The Abstract Factory, which we looked at earlier, would be inadequate for the task for several reasons. First, the Abstract Factory generally expects the configuration and settings to be injected when the Client commands the Factory to create the object. Second, the Abstract Factory assumes that creation is handled in a single function. Third, the Abstract Factory returns the objects created immediately.

The solution is instead to use a Builder object, which can manage the creation process as a series of steps, maintain the creation state, and return the created object as a final step. The Client manages the process by working closely with the Builder interface, providing the data needed for the various objects.




In this example, the Controller is responsible for constructing the response object through the Builder interface. The Builder is responsible for creating an appropriate response, whether that be an OkResponse (200), a PermanentRedirectResponse (302) or a NotFoundResponse (404). The steps to create the object are handled by the Builder, but the data is provided by the Controller.

The Builder has a few important parts. The first part is the Client — in this case, the Controller. The Client is integrally associated with the construction process and feeds information to the Builder as it becomes available. In our example, the Controller would provide response information that the ResponseBuilder would use to create a proper response.

We also have the Builders themselves. We have an Abstract Builder, which is the interface that the Client knows; this is the way in which the Client can send information to the Builder. We use an interface to ensure that the Client doesn't know the implementation details of the Builder. We want to program our Client to the interface of the Builder instead, so that the Client can use all the Builders we create. This makes creating new Builders easier later. Additionally, we have the concrete Builders, which are implementations of the Abstract Builder interface.

Finally, just like in the Abstract Factory Pattern, we have the Product. The Product is the object that the Builder is creating. The Builder is responsible for building the Product, and the Client asks for the Product after it knows the construction work has been completed.

This is a departure from the Abstract Factory model: instead of the new Product being returned immediately, the Client must explicitly ask for the Product object. The Builder needs to take this step, while the Abstract Factory does not, because the Builder produces the object in steps that are invoked by the Client. If the Builder returned the object under construction right away, it could potentially return an unfinished object to the Client. The Abstract Factory doesn't have this problem, since creation is complete by the time the factory methods

finish running, and the fully configured object is immediately made available to the Client.

## Avoiding tight coupling

It's really easy with the Builder Pattern to think, "I'll just skip the interface component altogether; it's not important since I only have one Builder." But this kind of thinking creates some serious problems, not the least of which is that it tightly couples the Client and the Builder together.

The interface is critical to ensure Client and the Builder are loosely coupled, because the Client and the Builder should be reusable by unrelated objects in the future. Relying extensively on the concrete Builder objects means that the Client must know how the Builders work; this defeats the purpose of abstracting away the object creation process to another subsystem entirely.

The Client should always know about and expect an Abstract Builder, even if that's just a PHP interface for the Builder. In fact, creating an Abstract Builder should be the first thing that you do before creating the concrete Builders.

## Sample code

Let's continue our example here of a Controller creating a response object. We're going to assume that the Builder is able to determine which kind of product to create.

```
<?php

class Controller {

    public function account() {

        if(!$this->request->getUser()
              ->isAuthenticated()) {
            $this->response->setRedirect('/');
            return $this->response->render();
        }

        $this->response
         ->setTemplate('account.html');
        $this->response->setTitle('My
Account');
        return $this->response->render();
    }
}
```

Our Controller here is using the Builder Pattern to construct the response details and allowing the Builder to determine which rendering to utilize. Now let's take a look at our builders:

```
<?php

interface AbstractResponseBuilder {

    public function setRedirect($url);

    public function setTemplate($template);

    public function setTitle($title);

    public function render();
```

```
}

class ResponseBuilder implements
AbstractResponseBuilder{

    protected $product;

    public function setRedirect($url) {
        $this->product =
          new 301RedirectResponse($url);
    }

    public function setTemplate($template) {
        if(!($this->product
          instanceof 200OKResponse)) {
            $this->product = new 200OKResponse;
        }

         $this->product-
>setTemplate($template);
    }

    public function setTitle($title) {
        if (!$this->product
          instanceof 200OKResponse)) {
            $this->product = new
200OKResponse;
        }

        $this->product->setTitle($title);
    }

    public function render() {
        if(!is_object($this->product)) {
            throw new
InvalidResponseException();
        }
```

```
            return $this->product;
        }

}
```

As you can see in this example, there is a single Builder, called ResponseBuilder, that determines which kind of response to provide. Of course, we could have different Builders that build different responses, but one Builder is fine, too. This Builder can create different kinds of responses and knows how to configure them. The Controller knows how to send the right instructions and understands the interface, but doesn't know or care about the various responses. This is exactly the way we want it.

## The Builder and the Abstract Factory together

It's possible for us to use these two creational patterns together. For example, imagine that we wanted to build SQL queries and we wanted to make it possible for us to use different kinds of databases, too. Coming up with a particular type of ORM might be fine with the Abstract Factory Pattern, but we would then use the Builder Pattern to construct our SQL queries.

```
<?php

interface AbstractORM {
```

```
    public function getInstance();

}


class MySQLOrmFactory implements AbstractORM {

    public function getInstance() {
        return new MySQLORM();
    }
}

class PostgresOrmFactory
     implements AbstractORM {

    public function getInstance() {
        return new PostgresORM();
    }
}

interface ORMBuilder {

    public function setTable($table);
    public function setColumns(
         array $columns = array());
    public function setWhere(
         array $where = array());
    public function setOrderBy(
         $orderBy, $ascDesc);
    public function setLimit($limit, start);
    public function addJoin($rawSQL);

    // and so on...

}
```

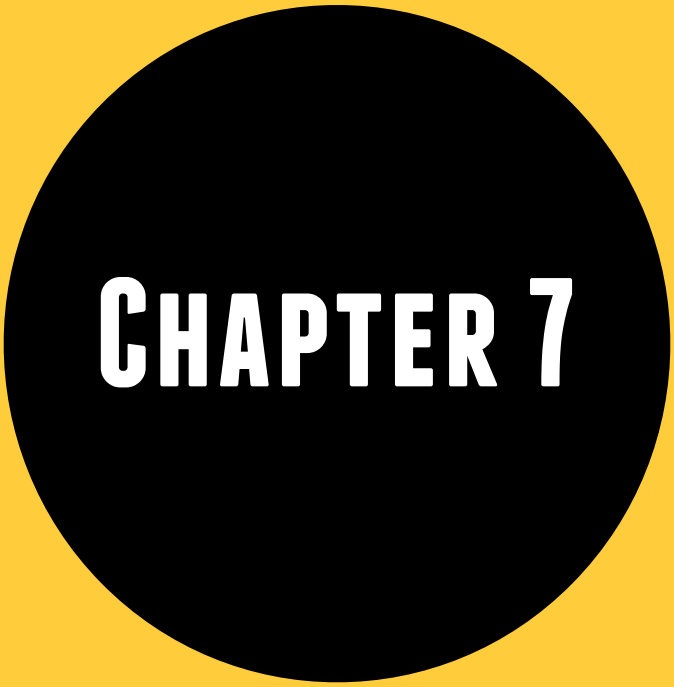
Here we have an Abstract Factory that is returning the Builder that we need in order to construct complex SQL queries for a particular type of database. It is possible to use these two patterns together, creating peace in our time.

## The Builder Pattern to the rescue

The Builder Pattern opens up a whole new world of options when it comes to creating complex objects within our applications. Instead of having to rely on instantiating the object on the outside or creating it within our Clients, we have the option of abstracting even the most complicated and difficult objects to something else. The Builder Pattern gives us new advantages and options in testing and developing our applications, and preserves most of the goals of SOLID application development.

Is the Builder Pattern more complicated than the Abstract Factory Pattern? Sure. But the complexity added here is more than compensated for by the power and flexibility we get back from the pattern itself. Some objects are going to be complex, and these complexities make a single creation function impossible. In those cases, the Builder Pattern comes to the rescue, giving us an option for solving even the most difficult creational challenges.

# CONQUERING THE LIMITATIONS OF INHERITANCE



# CONQUERING THE LIMITATIONS OF INHERITANCE

Imagine that you're developing the comments section of a website. You want to provide a few extensions that can filter text, like HTML and URLs. Your requirements call for the ability to filter out HTML, change URLs to a custom URL shortener, and replace emoticons with graphical representations. These should be configurable to be changeable at runtime.

As you start working, you build a foundation, and begin creating objects that inherit from that foundation to correctly handle the different use cases. But your boss comes to you and says, "What if our customer wants to change the URLs to a custom URL shortener AND replace emoticons with graphical representations?" And now all of a sudden, with standard inheritance, you're looking at creating a family of nine distinct objects to handle all possibilities. This is suboptimal, not to mention inefficient.

## Time to decorate!

PHP doesn't let us do multiple inheritance (traits are as close as we get, and it's not true multiple inheritance). Also, with

inheritance it's impossible to withdraw the responsibilities that we've given the object, unless we again extend the object and override that responsibility, which creates a hierarchical mess (and may affect other objects). And many times, it would be impractical to subclass anyway, since as we discussed, we don't want to have nine objects just to account for all the options available to us.

### Multiple Inheritance? What?

In object-oriented programming, inheritance allows us to create one class (a child) from another class (a parent), using the same implementation (or interface).

Multiple inheritance takes this idea a step further, by allowing two or more classes to form the foundation of a new class. Multiple inheritance is available in many languages like Perl, Python and Ruby.

PHP does not allow for multiple inheritance, though developers can implement multiple interfaces, and can also use multiple traits. We discuss traits in detail a bit later in this chapter.

There is a solution to this problem: it's called the Decorator Pattern. In its most basic form, the Decorator Pattern exists to allow us to dynamically add behaviors to our objects without having to rely upon standard inheritance schemes or create large families of objects with duplicate code. The Decorator Pattern is often portrayed as this monolithic, difficult, complex pattern, but in truth it's far simpler than that.

The Decorator components are fairly easy to understand. First, there's a common component interface: in this case, some kind of interface that defines the scope and behavior of the

component. This is implemented or extended to create the concrete instance of the component.

There may also be a separate interface for the Decorator itself, though this isn't explicitly required. Most of the time, using the interface for the component is straightforward enough, and the addition of the component as a property inside the decorator distinguishes the interfaces sufficiently.

Finally, we need a concrete Decorator that we can decorate the component with that we will define in any number of different ways.
Let's construct a simple example.

## String Decoration

Imagine that we want to create a string printing function. Our string printer will do something very basic: it will simply echo the string that it's given. Our interface, then, looks very simple and straightforward:

```
<?php
interface Printer {
    public function printString();
}
class StringPrinter implements Printer {
```

```
    protected $string;

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

    public function printString() {
        print($this->string);
    }

}
```

All we're doing here is taking a string and printing it. But we may want to do other things. For example, we might want to strip HTML from the string:

```
<?php

class StripTagsPrinter implements Printer {

    protected $printer;

    public function __construct(Printer
$printer) {
        $this->printer = $printer;
    }

    public function printString() {
        ob_start();
        $this->printer->printString();
        $string = ob_get_clean();
        print strip_tags($string);
    }
}
```

So now we have a decorator that we can use for removing tags from the string. We can also do a bunch of other things, like create a bold text decorator:

```
<?php

class BoldPrinter implements Printer {

    protected $printer;

    public function __construct(Printer
$printer) {
        $this->printer = $printer;
    }

    public function printString() {
        print '<strong>';
        $this->printer->printString();
        print '</strong>';
    }
}
```

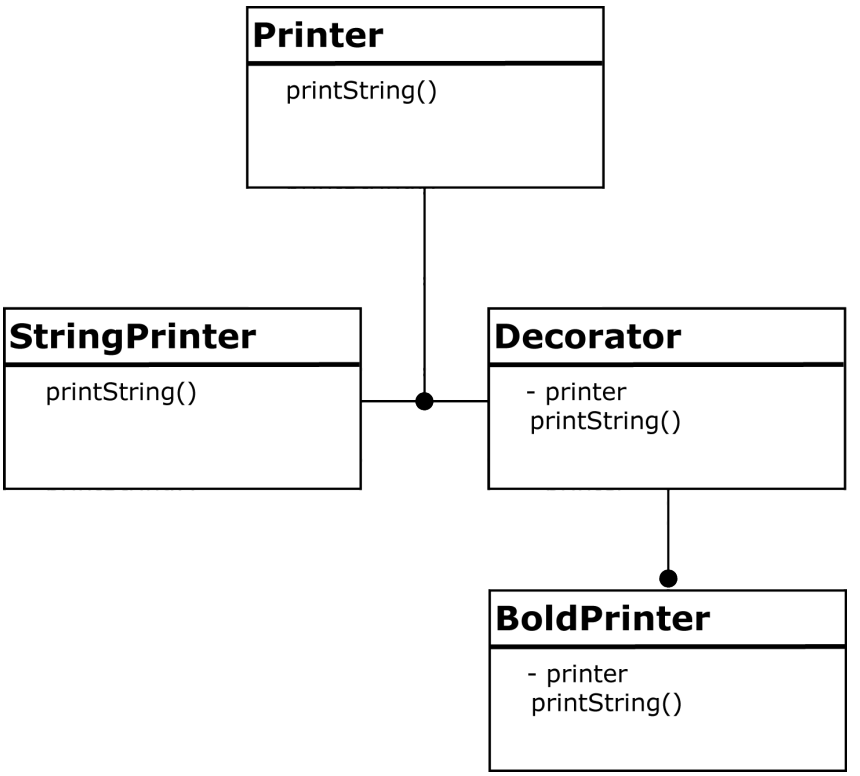
With our decorators, it's easy to create combinations that do precisely what we want, whether it's simply to print the string, or remove the tags, or remove the tags and bold what we get back:

```
<?php

$a = new StringPrinter('<br />hello world');
$a->printString(); // <br />hello world

$a = new StripTagsPrinter(
```

```
          new StringPrinter('<br />hello
world')
     );
$a->printString(); // hello world

$a = new BoldPrinter(
          new StripTagsPrinter(
               new StringPrinter(
                    '<br />hello world'
     )));
$a->printString(); //<strong> hello world
                   // </strong>
```

## Decorating for fun and profit

So why does this work? Let's draw a chart illustrating the architecture we've created here:



We have our component, also known as the Printer interface. This is implemented in a concrete object called StringPrinter. While many Decorator Pattern examples show a separate decorator interface or class, this isn't actually required to effectively develop decorators. In fact, we don't need that here (though you may find from time to time that it's necessary). Still, each and every one of our decorators extends from the Printer interface.

In some cases, you won't explicitly define a decorator interface. That's okay: when we're talking about decorators, we're talking about their behaviors; the chart shows an interface not because one is required, but because the interface is typically used to define a group of behaviors. A de facto interface is created by developing the decorators even if you don't spell one out. It helps to imagine that if you were to take your decorators and create an interface, what that interface would look like. Then, even if you don't establish a specific, explicit interface for your decorator, you will realize you've established a de facto interface simply by how the objects behave.

## Decorating with traits

Starting with PHP 5.4, it's possible to use traits in PHP. This allows us to actually decorate our classes with traits, instead of relying upon inheritance and wrapping to do the job.

Using traits requires a slightly different thought process from the previous (and traditional) decorator model. By using traits, we're still able to get the best advantages of the Decorator Pattern (namely the ability to reuse code, rather than rewriting it), along with a simple hierarchy. We also gain the advantages of a simpler object structure:

### What is a "trait?"

Starting with PHP 5.4, objects can express "traits", which are bits of code that can be mixed into objects.

These traits are NOT objects themselves (they cannot be directly instantiated), nor do they inform the object's type. But they do offer the ability to encapsulate bits of functionality in a way that is reusable across multiple objects.

```
<?php

interface Printer {

    public function printString();

}

trait StripTags {

    public function stripTags() {
        $this->string = strip_tags($this-
>string);
    }
}

trait BoldString {
```

```
    public function boldString() {
        $this->string = '<strong>' .
         $this->string . '</strong>';
    }
}

class PrintString implements Printer {

    protected $string;

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

    public function printString() {
        print $this->string;
    }

}

class BoldStringPrinter implements Printer {

    use BoldString;

    protected $string;

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

    public function printString() {
        $this->boldString();
        print $this->string;
    }
}

class BoldNoHtmlStringPrinter
```

```
     implements Printer {
    use StripTags, BoldString;

    protected $string;

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

    public function printString() {
        $this->stripTags();
        $this->boldString();
        print $this->string;
    }
}

$a = new PrintString('<br />hello world');
$a->printString(); // <br />hello world

$a = new BoldStringPrinter('<br />hello
world');
$a->printString(); // <strong><br />hello
world

$a = new BoldNoHtmlStringPrinter('<br />hello
world');
$a->printString(); // <strong>hello world
                   // </strong>
```

In this case, the decorators themselves aren't objects or interfaces; instead, including the traits and implementing them is the responsibility of the class that uses the traits. This has the advantage of stating what should have been explicit before: decorators cannot stand on their own without the concrete object that they decorate. In this case, because we

are using traits, the code is explicit in saying that the decorator is not a stand-alone object, but part of the decoration of other objects.

Another advantage here is also that these decorators can be used other places where the Printer interface is not used, thus freeing the traits for reuse by other parts of our application.

## The challenges and drawbacks of the Decorator

The Decorator is a wonderful pattern but it does have important drawbacks that we should be aware of.

The Decorator creates objects that implement a common interface (like the Printer interface in our example); however, depending on whether or not the object is the concrete component or the decorator, the objects behave differently. The StringPrinter takes a string as the argument to the constructor, while each decorator takes an object of type Printer. This is a subtle but important difference.

What this means is that the decorators should be created before sending them to objects that need them, to avoid chain construction problems. If you try to create one of these objects but you pass a string when an instance of Printer is expected, you'll run into a problem (specifically, a fatal error).

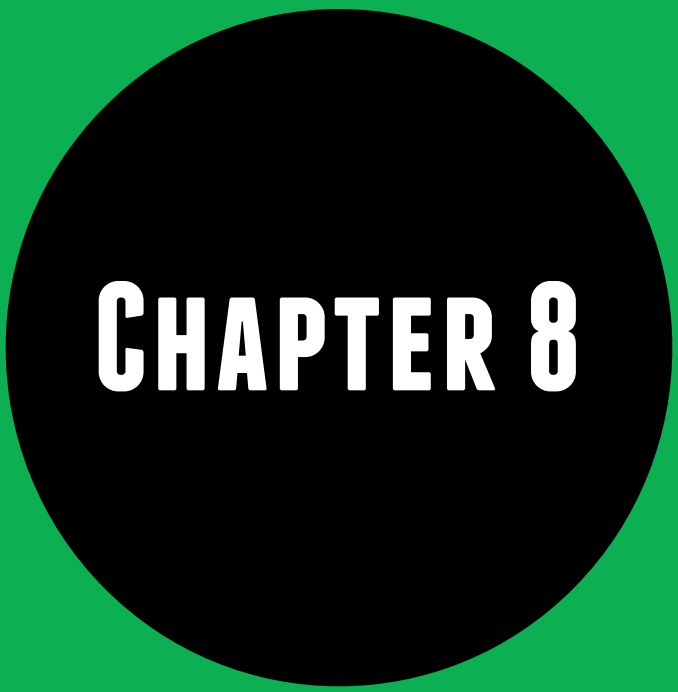
Also, because the Printer interface is universal for each of the objects, type hinting on the Printer interface may not give you exactly what you're looking for. It's important that your application understand the distinct differences between the objects and account for them in its handling of the decorators.

While traits can be useful in solving some of these issues (objects don't adopt types based on the traits they have), they come with their own set of problems. For example, traits obfuscate the behavior to some degree: traits become part of the object, and so it's not immediately obvious that the traits are included (without observing the code). The decoration isn't automatic with traits either; it requires that the developer specifically account for the traits. This means that the advantages of a common interface are lost when using traits, especially if the API of a given trait changes.

## A common, tremendously useful pattern

The Decorator Pattern is an old pattern that has been used thousands of times. You have probably come across it in your career, and if you haven't, you will. Decorators are common, powerful, useful and effective tools that, when used well, make development far easier for the average developer.

# TRANSLATING BETWEEN OBJECTS



# TRANSLATING BETWEEN OBJECTS

The challenge of getting different objects to communicate in an effective, simple way is one of the hardest problems to solve in object-oriented development. There are so many tradeoffs and pitfalls that object communication is often where applications live or die. Many developers don't realize until it's too late that creating solid, cohesive communication techniques is their chief responsibility.

There are three design patterns that focus exclusively on helping different objects in different parts of an application communicate with one another. These three patterns have different intents, but they all seek to answer a common question: how can I decouple objects from each other, give them a way to communicate, and still allow them to do their individual work?

The Adapter Pattern provides a solution by taking two or more common objects with unique interfaces and creating a single, common interface that can be used by other objects. The Adapter Pattern is usually implemented after the different

objects are implemented, or when joining different libraries designed by different developers.

The Bridge Pattern seeks to create a unified interface which talks to a common implementation interface. The Bridge Pattern is used before objects exist, allowing developers the opportunity to plan, create, and consider their requirements and options.

Finally, the Facade Pattern offers up a solution for the creation of an interface for a set of objects. The Facade Pattern might seem to be very similar to the Adapter Pattern, but there's a distinct difference: the Facade Pattern creates a new interface where one didn't previously exist, while the Adapter Pattern is designed to make two existing interfaces work together transparently.

## Three patterns, three different use cases

These three options can look and feel very similar, but there are important differences between each of these patterns that are important to remember, recognize and apply.

The use cases for the Adapter Pattern and the Bridge Pattern are essentially the same: to abstract the implementation of a system from the components making use of the implementation through the common interface. But the Bridge Pattern is implemented when you are designing the

subsystems. The level of control you have over the subsystems allows you to both design the implementation and the interface. Recognizing that the subsystem may grow and change is a crucial part of understanding the need for the Bridge Pattern. In fact, many if not most of the refactoring challenges faced in object-oriented applications can be solved by proper use of this pattern.

The Adapter Pattern, on the other hand, is a pattern that is largely used when you did not design the subsystems for which you're defining the interface. The Adapter Pattern allows you to make two or more distinct interfaces work together through a common interface. Different data storage layers or caching mechanisms often have distinct, disparate interfaces that you had no hand in designing, but the Adapter Pattern gives you an opportunity to standardize the interface for your specific application.

Many people often confuse the Facade Pattern and the Adapter Pattern, but they are unique and distinct as well. The Facade Pattern is used to create an interface where no common interface existed before. This is a subtle but important difference: the different caching libraries have different interfaces, but they do have commonalities (such as getting, setting and deleting keys) that make it possible to create a common interface between them. Different, unrelated subsystems, on the other hand, often don't have such

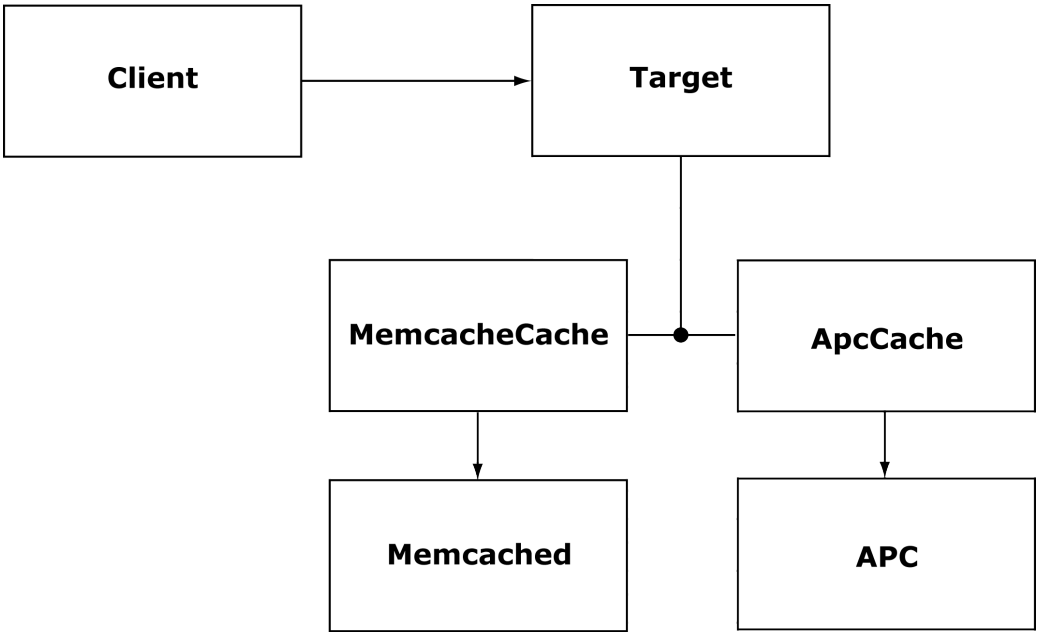
commonalities; the Facade Pattern, therefore, is an attempt to create a new interface to bring these systems together in a unified fashion. The Facade is generally tightly coupled to the subsystems it represents, and creates a completely new interface where none previously existed to abstract the subsystem from higher level components.

## Unifying two disparate interfaces with the Adapter Pattern

Lots of times, developers come across two different objects that have different interfaces but need to either work together or work interchangeably with other objects. As a result, the developer creates an adapter. This is the purpose of the adapter: to take different but similar objects and create a common, unifying interface that allows them to be swapped out for one another.

Adapters are so common and useful that you may have implemented one before and not realized it. There are four components to the adapter. The first is the Client; this is the object that actually talks with the adapter. There's also a Target, which is a defined interface for the adapter and is the interface that the Client understands and expects. The Adapter itself implements the Target interface and holds a reference to the Adaptee. The Adaptee is the object that you are actually adapting for.

When we discussed the Abstract Factory Pattern, we talked about the creation of different types of caches. Let's examine how the Adapter Pattern can be used to create different types of caches while creating a common, uniform interface.




First, we define an interface for our cache:

```
<?php
interface Cache {
    public function get($key);
    public function set($key, $value, $ttl);
    public function delete($key);
    public function purge();
}
```

The interface is simple enough: we can get and set cache items, delete a specific key, and purge the entire cache. Implementing this is relatively simple and straightforward for both Memcache and APC:

```
<?php

class MemcacheCache implements Cache {

    protected $memcache;

    public function __construct(array
$servers) {

        $this->memcache = new Memcached();

        foreach($servers as $host => $port) {
            $this->memcache
              ->addServer($host, $port);
        }
    }

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

    public function set($key,
                        $value,
                        $ttl = 3600) {
        $this->memcache->set($key, $value,
$ttl);
    }

    public function delete($key) {
```

```
        return $this->memcache->delete($key);
    }

    public function purge() {
        return $this->memcache->flush();
    }
}

class ApcCache implements Cache {

    public function get($key) {
        return apc_fetch($key);
    }

    public function set($key, $value, $ttl =
3600) {
        return apc_store($key, $value, $ttl);
    }

    public function delete($key) {
        return apc_delete($key);
    }

    public function purge() {
        return apc_clear_cache("user");
    }
}
```

These two cache objects are interchangeable with one another without so much as changing a line of code. If the cache is set by a configuration variable, the cache can be set and changed at runtime.

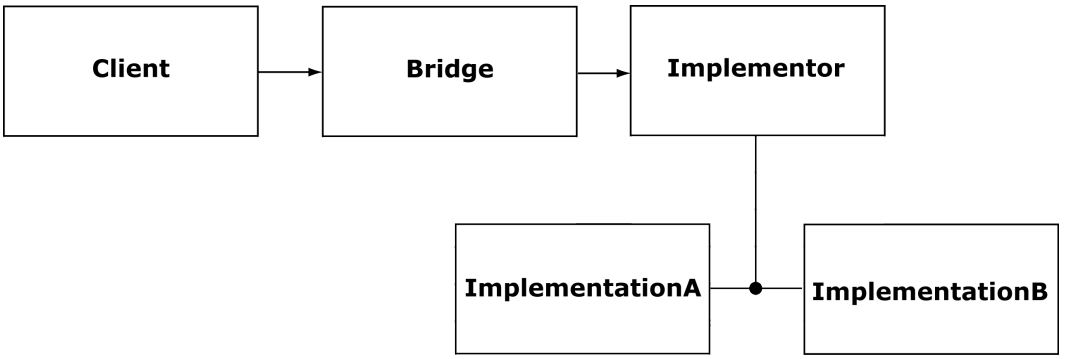
In this way, the Adapter Pattern is specifically designed for honoring the Liskov Substitution Principle, because one

subtype can be swapped for another. The tight coupling between the Adapter and the Adaptee might create some concerns for many object-oriented developers, but this tight coupling ensures that the Adapter and the Client can be loosely coupled together through the Target interface

## Designing the interface and the implementation

The Adapter Pattern is the go-to pattern for creating an interface for objects that already exist. But when developers are creating both the interface and the implementation, they have an opportunity to define both, creating additional consistency. This is the use case of the Bridge Pattern.

The Bridge Pattern lets us define an interface that interacts with a specific set of concrete implementations. The reason that we use the interface at all is that we want to be able to make changes in a single object as the implementations grow or change.



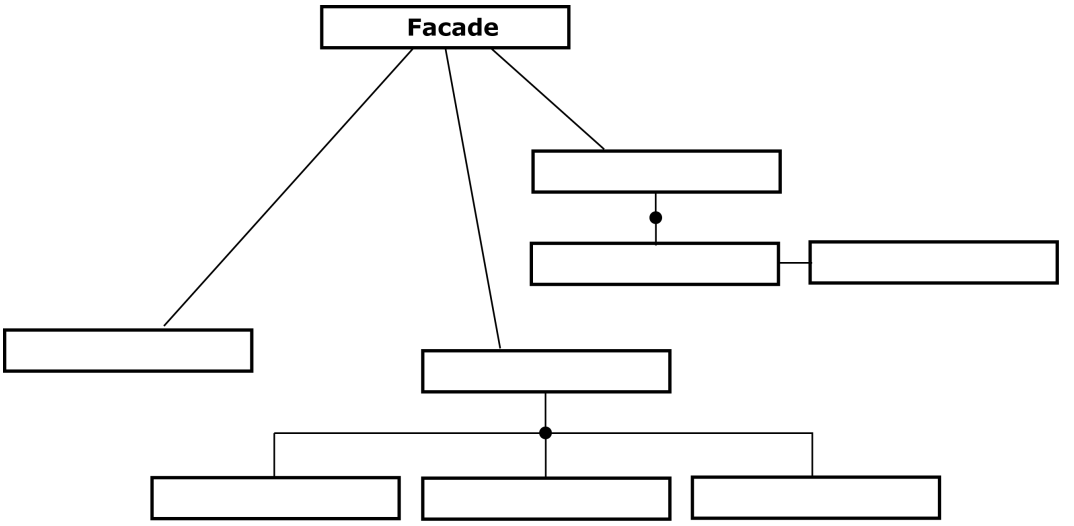
As the implementation changes, so does the bridge. This allows the developer to abstract the change process away from the objects that use the Bridge interface, while still allowing for growth of the Implementor class and its subclasses. Also, since the Bridge can be subclassed to support new features that require an interface adjustment, the Bridge pattern helps account for future development. The decoupled nature of the client from the implementation also allows for swapping the Bridge and the Implementor out at runtime without having to change the code of the Client.

## Creating an interface where one didn't exist before

Occasionally developers have a group of objects that form a subsystem or act in concert with one another. The Facade Pattern helps us create an interface to work with a group of objects where no common interface existed before.

The Facade Pattern requires a tradeoff of certain object-oriented principles (loose coupling, single responsibility principle) for the benefit of simplicity in higher-level systems. The Facade Pattern abstracts away the need to understand how to interface with sub-components. Because the Facade itself is responsible for this communication, the objects that use the Facade don't need to be aware of the subsystem. This improves the loose coupling of higher-level systems by

ensuring that they don't focus at all on the subsystems they need.




The Facade Pattern ties different (and often unrelated) sub-components together into a single component that can be used at a higher level. It's useful for creating a simple interface to a more complex subsystem, or in layering subsystems throughout your application.

Many people think that a Facade should be a Singleton because you often only need a single Facade. However, controlling the number of Facades is easily accomplished by careful programming and doesn't need to be enforced through programmatic means.

The Facade Pattern often implements the Abstract Factory as a way of creating the subsystem objects that are needed by the Facade. This is a good strategy for abstracting the creation process, but it's important to be careful not to over-architect the Facade. Over-architecting the Facade (by giving it additional work to do) results in a Facade that is overly complicated, and that abstracts too much of the work away from both the higher-level and lower-level objects. While we should expect that certain rules are violated by the Facade (like rules on tight coupling and single responsibility), these violations should improve readability or simplicity of the application as a whole, rather than create additional challenges for developers.

## Similar patterns, different uses

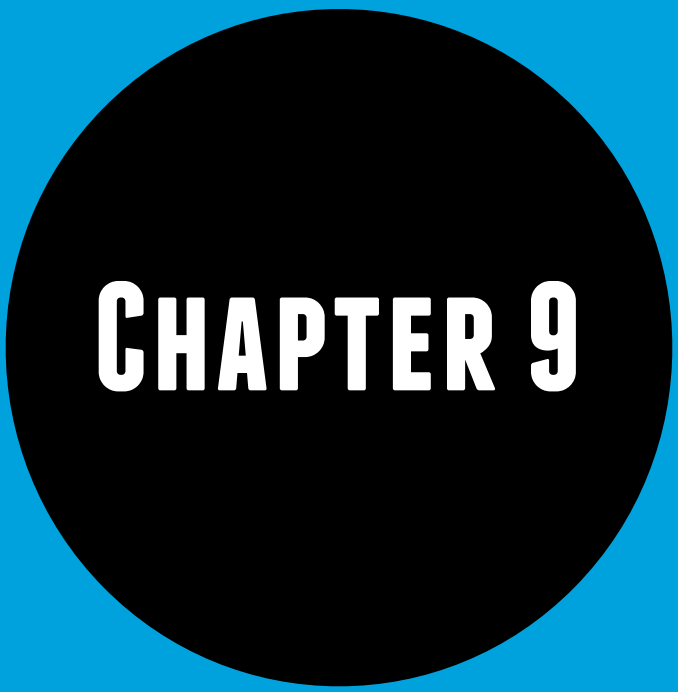
These patterns look extremely similar to one another, and they differ more on why they are implemented than on how. In fact, implementing an Adapter can look almost identical to implementing a Facade. Even if two objects look similar in their implementation, the reason for their implementation is the factor that controls which pattern you have used, rather than their specific code or structure.

Even though these patterns are similar, understanding all three is the key to ultimately being successful with each of them. And even though the Bridge Pattern is the least common of the three (since most of us never get to design the library

and the way it will be used), it's one of the most powerful patterns when it's used correctly.

With these three patterns you can solve a multitude of problems. They don't solve everything but they make life easier in so many different ways.

# A GOOD STRATEGY...



# A GOOD STRATEGY...

There are many times in application development where, depending on the conditions at runtime, different algorithms come into play. For example, depending on the type of data coming into a system, you might want to change exactly what kind of validation logic is applied. There are a number of ways to do this, ranging from simple to complex. In procedural programming, you could use a switch or a complex set of conditionals (if-else statements). Or, in object-oriented programming, you can use the Strategy Pattern.

The Strategy Pattern makes it possible to define a common interface for distinct algorithms that perform different behaviors. For example, you can define a strategy that knows how to handle users' first and last names differently from their email addresses. Depending on which data type is detected, a different validator is used, but the application is still capable of validating both. This strategy would accept the same arguments (the passed data) but would perform different algorithmic checks on the incoming data.

The Strategy Pattern is very popular, especially in frameworks and libraries, but has many other uses.

## A good plan of attack

Suppose that you know you're going to be getting either JSON or XML passed into your application via the API. Since the only two valid options are JSON or XML, while all others are invalid, you need only to implement a strategy for handling these two formats. You also know that what you want to do is convert each of these formats into a three-dimensional array that can be passed around, iterated on and handled. Your job, then, is to construct a seamless way for the Context to identify which type of data has come in, handle it, and return results that the application can utilize.

The Strategy Pattern is your friend in this case. Regardless of whether you are getting JSON or XML, you can still plan to hand the various data types to a commonly known interface. In turn, that interface can handle the data based on precisely the type of data you've been given.

Without the Strategy Pattern, we would encapsulate the data processing within the application, like this:

```
<?php

class HandleApiData {
```

```
    public function processData($data) {

        switch($this->identifyDataType($data))
{

            case 'json':
                // do some JSON handling
                break;

            case 'xml':
                // do some XML handling
                break;

            default:
                throw new
InvalidArgumentException();
                break;
        }
        return $processedData;
    }

}
```

While processing the JSON data is a simple, single function call, processing XML can be involved and difficult. As a result, this function is liable to be quite long. Also, the Single Responsibility Principle would tell us that we're doing too much in this class and it should be broken up.

Enter the Strategy Pattern. We can use Strategy to create a common interface, and then process the data inside the Strategy class we created.

```php
<?php

interface DataStrategy {

    public function handleData($data);

}

class JsonDataStrategy implements DataStrategy
{

    public function handleData($data) {
        return json_decode($data, true);
    }

}

class HandleApiData {

    protected function identifyDataType($data)
{
        if($this->xml($data)) {
            return new XmlDataStrategy();
        } elseif ($this->json($data)) {
            return new JsonDataStrategy();
        }

        throw new InvalidArgumentException();
    }

    public function processData($data) {

        $strategy = $this-
>identifyDataType($data);
        return $strategy->handleData($data);
    }
}
```

While we haven't defined a class that handles the XML data in our example, you can imagine what that class would look like. If the HandleApiData class detects that the data is in fact JSON or XML, it instantiates the correct strategy, which it then returns to the processData() method. This method then invokes the Strategy API and returns the results. The JsonDataStrategy class handles the JSON data correctly and returns it, but the HandleApiData method doesn't have to know how this is accomplished.

The Strategy Pattern opens up a whole host of new options for us in terms of handling different types of data without the handling class (the Context) having to know exactly what methodology was used.

## The elephant in the room

There are a couple of pretty serious objections to this pattern that often pop up. We've traded one set of responsibilities for another: handling the data and creating the strategy objects when needed.

We've also tightly coupled the HandleApiData object context to the various strategy objects we've created. Since the controller in this case directly instantiates the objects it needs, they are, by definition, tightly coupled.

But there is an important and worthwhile distinction to be made here. First, the HandleApiData object does very little of the actual work. It's more a coordinator for the various Strategy objects, which handle most of the responsibilities. Second, even though we are tightly coupled to the various class names for the concrete instances, we are still really only tying ourselves to the interface we defined in DataStrategy. This is because we are not expecting the strategy objects to have additional methods, and thus we care not about their implementations in the Context.

Of course, this becomes less true as the complexity of the strategy increases. In fact, in extremely complex strategies, the Context may have to know a great deal about the implementation of the strategy in order for it to properly select the right strategy for the job. However, we are trading the tightly coupled relationship for the encapsulation of various algorithmic behaviors, which are often much more complex than the simple Context object.

## Passing data to the Strategy

There are a few different ways to get data into the Strategy. We've discussed one: directly passing in the data to the Strategy. The advantage here is that the Strategy is essentially stateless, and the Strategy doesn't have to know anything about the Context, which decouples the Strategy.

Another option is passing the Context to the Strategy as an argument. This has the advantage of making the entire Context available to the Strategy based on what the Strategy needs (useful for complex Context/Strategy relationships), but adds a drawback: the Strategy is now tightly coupled to the Context, and the Strategy becomes an object that has state. By giving the Strategy state, it cannot be shared with multiple Contexts (you must instantiate one strategy per context in this case).

### WHY PASS THE CONTEXT TO THE STRATEGY?

There may be times when the Strategy you're implementing is complex, or needs to possibly query the Context for additional information. In those cases, it's useful for the Strategy to interrogate the Context, and it must have a reference to the Context to do so.

For example, imagine that the Strategy is handling HTML tags. The ability to page ahead inside the Context and identify if the tag is closed, or read further ahead in the document, gives the Strategy valuable information. This would be a case to pass the Context into the Strategy.

## Uses for the Strategy Pattern

The Strategy Pattern has many uses. In fact, it's one of the most flexible patterns out there if you can accept the tradeoffs.

The Strategy Pattern is used in many cases: breaking, tokenizing and representing HTML, filtering different kinds of data, and allowing for different output options, just to name a

few. The beauty of the Strategy Pattern is that it offers us the ability to create objects that know nothing about how they'll be used, but that can be used in a variety of different ways. This flexibility offers a developer the opportunity to configure and develop an application that suits their needs and tastes without having to rewrite or redevelop tremendous amounts of code. Giving developers the opportunity to customize their applications with ease and simplicity is just a good strategy.

# AN OBJECT BETWEEN FRIENDS

# AN OBJECT BETWEEN FRIENDS

As an application grows, its complexity and number of objects are likely to grow with it. In fact, each new feature will probably be encapsulated by an object or a family of objects, meaning that each new feature adds a bunch of new objects that other objects may have to know about and work with.

Given that we want to encourage reuse and loose coupling in our objects, sometimes it becomes necessary to create a way to handle these objects independently of one another, even when the objects have to work together.

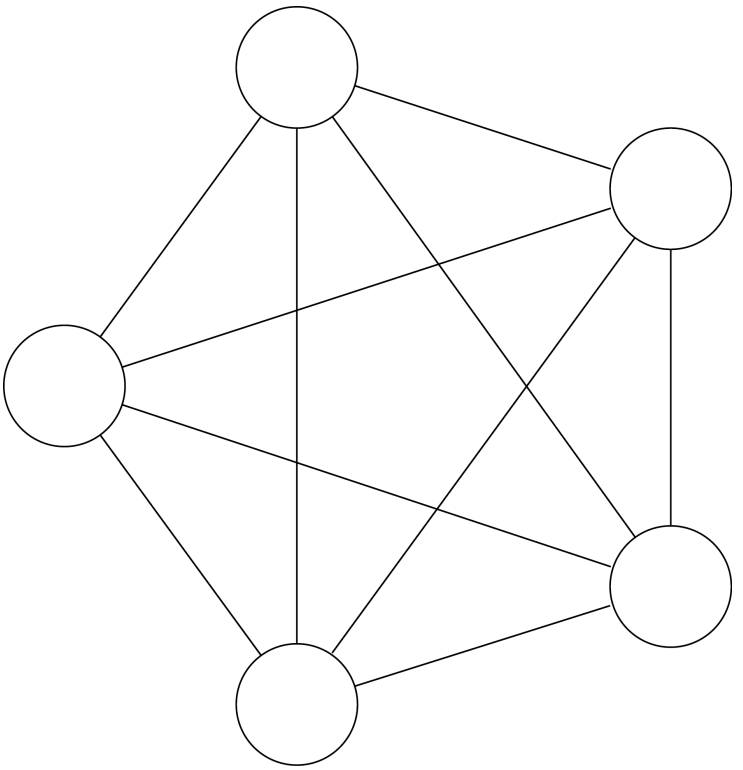
The problem is that as the number of objects grows, so does their interdependencies. Interdependencies result in tight coupling: objects that have to understand the innerworkings of one another to some extent; as a result, it's easy to end up in a situation where it's impossible to remove one or more objects from the group and use them in another way, because they are so dependent on the other objects in the group. This also creates a maintenance nightmare, since one change in an

object can have an impact across a wide range of other objects.

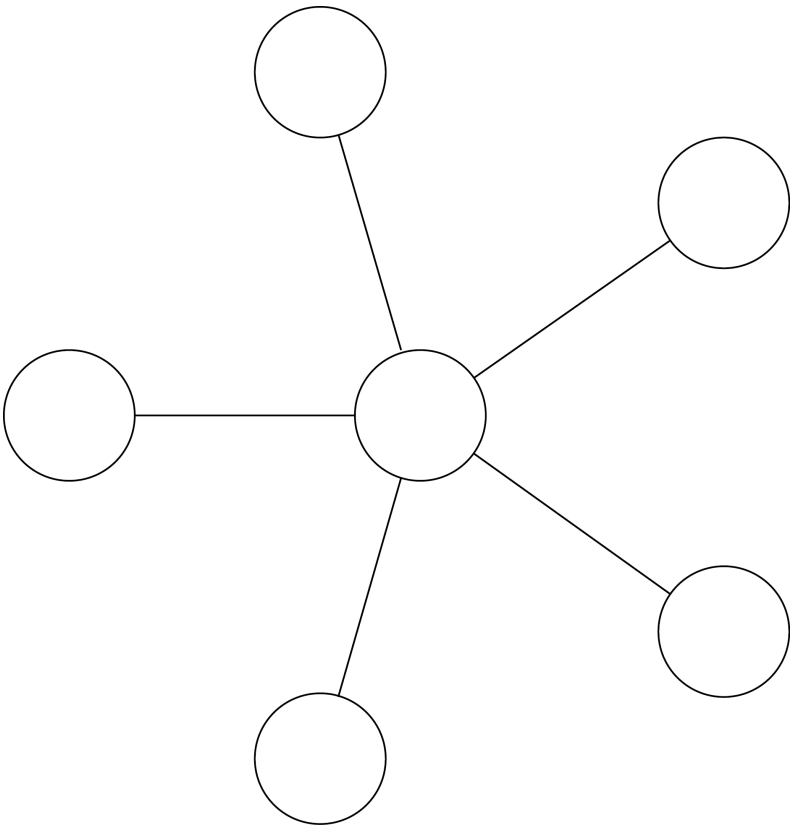
The solution to this problem is to make use of something called the Mediator Pattern.

## Centralizing tight coupling

One of the biggest challenges that objects face is the possibility that they might have to know about the implementation of other objects. In fact, this is the definition of tight coupling: having two (or more) objects that depend on one another's specific implementation (rather than relying solely on each other's interfaces).



The Mediator Pattern helps us to solve this issue. We remove the problem of having a group of objects that depend upon one another by creating a centralized Mediator. The Mediator is responsible for facilitating communication between the various colleague options. The colleagues in turn can focus on their individual responsibilities, and rely on the Mediator to handle all communications. This reduces coupling and increases flexibility.



The Mediator has three responsibilities:

- Know the interfaces of all the objects for which it mediates
- Abstract away the communication between objects so that objects do not have to communicate directly with one another
- Remove the need for objects to know about other objects within the system, encouraging their reuse

## Pass the salt, please

Since Mediators facilitate communication between other objects, it's important to focus on what Mediators look like, as well as on the behaviors you'll want to avoid when crafting a Mediator object.

It can be tempting to have a Mediator object determine how to handle a request, validate data, or do other "business logic." This is outside the scope of the Mediator. The Gang of Four calls the Mediator pattern "an object that encapsulates how a set of objects interact"; this definition leaves little room for a Mediator object to do business logic or data validation. These behaviors belong in the objects that the Mediator communicates with.

Mediators can come in all shapes and sizes; in fact, most applications use a Mediator pattern in the Web world. We call it Model-View-Controller, but it's essentially the Mediator pattern: the Controller mediates between the View and the

Model, ensuring that those two components can work independently and without knowing the internals or operations of the other. Most models and views don't even know that the other component exists, and that's perfectly acceptable. The Mediator is the only object that really cares.

Another common type of Mediator is domain modeling. In domain modeling (covered in "Beautiful Models"), the Value Object represents the data while the Data Storage layer is responsible for saving and fetching data from various data stores. These two objects never speak to one another directly; the Mediator instead communicates with both, offering some abstraction between the representation of the data and how the data is ultimately persisted between requests.

## How is this not just creating an Adapter/ Facade?

It's true: this pattern looks very similar to the Adapter and Facade patterns we just discussed. In fact, it might feel safe to assume that the Mediator is just a variation on one or both of these patterns. But that assumption would be wrong.

The key difference is in exactly what you're trying to accomplish. The Adapter Pattern is focused on translating one interface into another. The Facade Pattern is attempting to create a new interface where none existed before. But the Mediator Pattern is different: it's focused on facilitating communication between colleagues — objects that are on the

same level — and not on creating an interface for objects to use.

Thus, when the Client is involved with the Mediator, it becomes a colleague to the components that it is talking with. The Model and the View are essentially equals with different purposes in Model-View-Controller. This is different from an Adapter or a Facade, where the cache details are abstracted away and are not colleagues with the objects that utilize them. This may appear to be a distinction without a difference. However, there's an important difference: colleagues are equals, while adaptees and subsystems are hierarchically lower than the objects that interface them. Understanding and implementing this principle is a crucial component in getting your implementation correct; failing to treat each object in the Mediator pattern as an equal with every other object will result in a Mediator that ultimately doesn't accomplish your goals. And you'll find that it's increasingly difficult to implement the Mediator without doing business logic or validation in the Mediator itself, a definite violation of the goals.

## Sample code

The Mediator is a simple object, usually facilitating communication by receiving messages and passing them along to others. Our illustration will be the most common Mediator you're likely to encounter: the Controller in a Model-View-Controller application. This Controller's job will be to take

a POST request from an API, send that data to be processed and saved, and return a response as appropriate.

```
<?php

namespace Application\Controllers;

class User {

    public function createUser() {
        $apiKey = $this->request-
>getPost('apikey');
        if(!$this->validateApiKey($apiKey)) {
            return new 401NotAuthorized('not
authorized');
        }

        $payload = $this->request-
>getRawPost();

        try {
            $results = $model-
>createUser($payload);
        } catch (ModelException $e) {
            return new 400BadRequest($e-
>getMessage());
        }

        return new 200OK($results);
    }
}
```

Let's examine what's going on here more closely. The Controller (Application\Controllers\User) is responsible for mediating between a variety of objects. First, it gets data out of the Request object. It sends that data to itself for validation,

and then sends it to the model for processing. Finally, it's responsible for returning a response object. Since we're writing this as an API, we're returning response objects that follow the specific HTTP status codes we need, but returning an array or a response object for consumption by a view layer would be just as effective.

## Aren't I doing this already?

This might not seem to be that revolutionary. It's not. In fact, most of us do this on a regular basis without even thinking about it. Remember my statement that design patterns are common solutions to common problems? The Mediator Pattern is one of the best illustrations of this: it's so natural that we do it automatically, without even thinking about it. This is the power of design patterns: well-constructed patterns come naturally, occur naturally, and happen without anyone thinking too hard about them.

The best thing about the Mediator Pattern is that it's really easy to start using right away. It doesn't require any special training or thought; decoupling objects from one another and implementing a Mediator is a simple, quick refactoring technique you can use to improve your applications immediately. When you are confronted with objects that have numerous references to other objects, this situation is a great candidate for implementing the Mediator Pattern. Once the Mediator Pattern is in place and functioning, a whole host of

other actions become much easier, like writing unit tests or developing subclasses of your objects. Since the Mediator removes the dependencies through abstraction, your focus can be on solving whatever problem a particular object has, without worrying about the other related objects.

# LET'S TALK

## CHAPTER 11

# LET'S TALK

From time to time you'll run across cases where objects need to alert other objects about certain changes in their state. You might want to log messages to the system log, for example, or update related value objects.

There are many different ways to alert objects about state changes in other objects, and my personal favorite is known as the Observer Pattern.

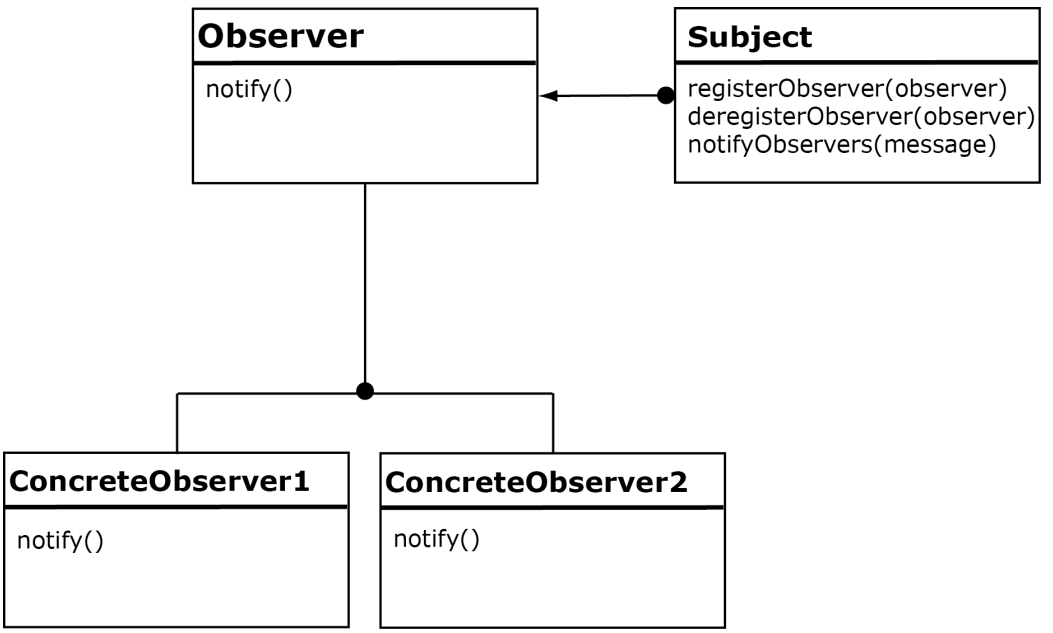
Even though this pattern is far more common in applications that have windows or in real-time applications (think Javascript), it still has a value in web-based PHP applications. In fact, its value in PHP applications is often underestimated by people who see it as an old relic of a time when objects didn't die at the end of the request cycle. But the power and simplicity of this pattern makes it attractive even in share-nothing situations like PHP.

## Tell me about your changes

The whole point of the Observer Pattern is to alert a group of objects when one object changes state in some prescribed way. As such, while the Observer Pattern is a pattern in and of itself, it's often implemented in conjunction with other patterns. In fact, there are few times you'll see the Observer Pattern implemented by itself.

The objects interact with one another in a fashion similar to this diagram:

This pattern has two components: the Publisher and the Subscriber. Each of these implements an interface, which identifies the way in which the objects communicate. Most of the time, the Observer Pattern is implemented as a one-to-many relationship between publishers and subscribers.
An interface for the Observer Pattern might look something like this:



```
<?php

interface Subject {

    public function registerObserver(
     Observer $subscriber);

    public function unregisterObserver(
     Observer $subscriber);

    public function notifyObservers($message);

}

interface Observer {

    public function notify($message);

}
```

An object that wants to publish messages would implement the Publisher interface, while an object observing other objects would implement the Subscriber interface.

When implementing the Publisher, it's important to permit more than one Subscriber to be subscribed at any given time. The process for subscribing objects is an implementation detail of the concrete class, and the interface doesn't have to worry about the details. Similarly, the Subscriber should have a reference to the object it observes, so that it can deregister itself as necessary; when allowing objects to unsubscribe, both Publishers and Subscribers should be able to properly deal

with the dependency chain they've created through the circular reference.

## What do I do with your data? Wouldn't you like to know!

The Observer Pattern offers several advantages over other schemes for sharing state changes, and the biggest one is the abstraction of the subscriber from the publisher itself.

Once the message is sent off to the subscriber, the publisher doesn't know or care what happens to the data. The subscriber could ignore the data, log it, analyze it, or do any number of things with the message, but that's totally up to the subscriber. The publisher never needs to know what happened to the message.

## Pitfalls to avoid with the Observer Pattern

It's important to avoid some common pitfalls that can happen when using this particular pattern.

The Observer Pattern implements a circular reference between the subscribers and the publisher. The publisher holds a reference to the subscribers, and the subscribers each hold a reference to the publisher. This makes the process of cloning or serializing objects particularly challenging in many cases. Developers should take care to make sure that unserializable

or un-cloneable objects are handled properly, to avoid creating potential problems down the road.

It's also possible in cases where there are many updates being made to a particular Publisher that the resulting cascade of updates can be overwhelming to the system or create unnecessary noise. Especially in cases where the Subscriber might have a dependency on another subsystem (e.g. file system or database), it's important to consider the ramifications of the update process with this pattern.

Many developers attempt to resolve this particular concern and end up making one of two other common mistakes by implementing either the push model or the pull model. Both of these models add additional drawbacks and create new challenges that have to be addressed, and are really not meant to be part of this pattern at all.

In the push model, the Publisher makes a decision about what to send to the Subject, and sends only data that the Publisher believes the Subject will want. However, this is not the role of the Publisher; the Publisher's job is to send messages to all subscribers equally. It's not the job of the Publisher to determine what a particular Subscriber might want or can handle; instead, each Subscriber should receive the exact same notification, and the Subscriber is responsible for handling that notification in an appropriate fashion. In fact,

beyond knowing that the Publisher has subscribers, the Publisher shouldn't inspect or care about the specific types or interests of the Subscribers; as a result, it must treat them all equally.

Other times, developers will decide to send only a token message about an update and allow each Subscriber to inspect the Publisher for changes that might interest the Subscriber. This results in a new set of challenges. The Publisher should be sending complete messages to all Subscribers when a change takes place, rather than a token message that a change has occurred. Requiring the Subscribers to inspect the Publisher requires that the Subscriber know the internals of the Publisher in order to know where to look. This defeats the purpose of abstracting the Publisher from the Subscriber.

In applications that might allow multiple updates to occur within the Publisher in a short period of time, or in applications which run constantly (e.g. on the command line), it's important to consider and avoid race conditions in the notification process.

## The Observer vs. the Mediator

The Observer Pattern is often confused with the Mediator Pattern, especially given that one Publisher sends messages to multiple Subscribers.

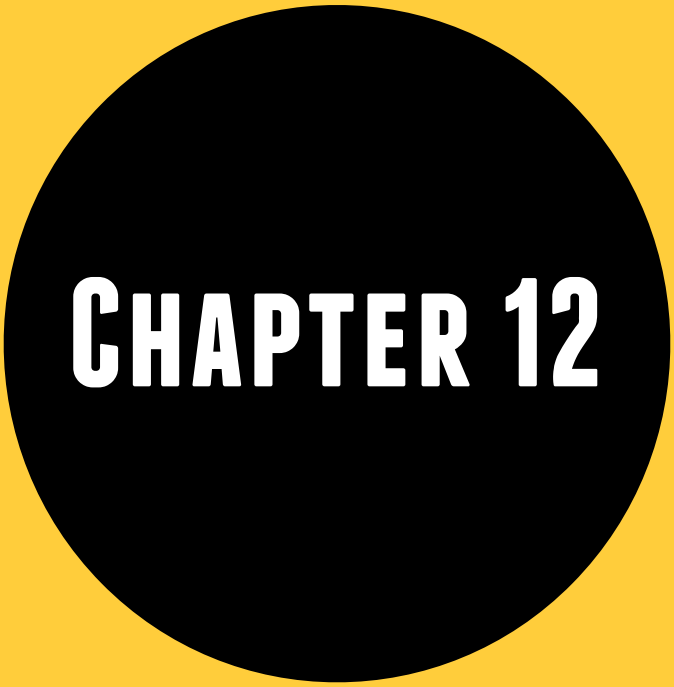
However, the difference between the two patterns is that the Mediator Pattern mediates between colleagues, while the Publisher is an object that notifies outside objects that are not necessarily colleagues of the Publisher.

The Mediator Pattern expects that objects will pass messages back to it and inform it of their behavior. In the Observer Pattern, the purpose and intention of the Subscriber is unknown to the Publisher, and the Subscriber doesn't send messages back other than to acknowledge the receipt of the notification. Additionally, the Publisher doesn't manage communication between the Subscribers, which may or may not know about each other.

## For more than real-time applications

Even though the Observer Pattern is common in real-time applications, it has incredible value in PHP applications as well. It's easy to write off this pattern as a relic or as something that only needs to exist in Javascript components. However, I find that being able to notify other objects of important state changes dramatically improves the overall quality of my applications, not to mention my sanity. Logging becomes much easier when the log doesn't have to know the objects that call it, which is the beauty and simplicity of the Observer Pattern.

# WHOSE LINE IS IT ANYWAY?



# WHOSE LINE IS IT ANYWAY?

Like every good PHP developer, you're writing your own framework. You know that your framework has to be flexible. To accomplish this, you're going to route requests based on a set of regular expressions defined in the config. You'll handle each of the regular expressions in the order they're defined, and you want to give them special properties like default arguments. You expect that each regular expression will be converted into an object, but you're not certain how to solve this particular problem.

Thankfully, there's a pattern for that. It's called the Chain of Responsibility.

The Chain of Responsibility solves this problem in two ways. First, it abstracts away the process of handling the task to the various objects in the chain. Second, it allows an infinite number of objects to potentially handle the request based on their implementation.

The Chain of Responsibility works by defining a common interface, and allowing each object in the chain to determine if

it can handle the request. If it can't handle the request, it's responsible for passing it on to the next object in the chain. Each object has essentially two required roles: to handle the request and passing it along to the next object in the chain. This simple pattern offers a tremendous amount of flexibility for developers, especially framework developers.

## Creating the chain

The basic needs of the Chain of Responsibility are simple. First, you have to define some kind of common Handler interface so that your Clients know how to send their requests. This can be simple or complex; it's really dependent on your use case.

Once you've defined a Handler interface, you can start creating the Concrete Handlers that will actually handle the requests. Each Concrete Handler must do two things. First, it must implement the Handler interface (for consistency in the Client). Second, each Concrete Handler must have a way to pass the message along to the next Concrete Handler in the chain. If the Concrete Handler is the last in the chain, it should know how to tell the Client that the request couldn't be completed (there may be cases where the request falls off the chain, but letting this happen is generally poor form and we will discuss this in detail later).

It can be tempting to implement this pattern with a central Control object using the Mediator Pattern. This isn't how the Chain of Responsibility is designed. Each object in the Chain of Responsibility is able to handle the request like any other object; as a result, central control objects are discouraged.

## Implementing the Chain of Responsibility

Imagine that you want to create a set of email filters for your application. You can use the Chain of Responsibility to chain together email filters. The emails will run through each of the filters, and for a filter that matches the criteria established, we will handle that particular message in the defined way. It's also possible that an email doesn't match any filters, and you want to allow for the possibility that the email is not handled at all.

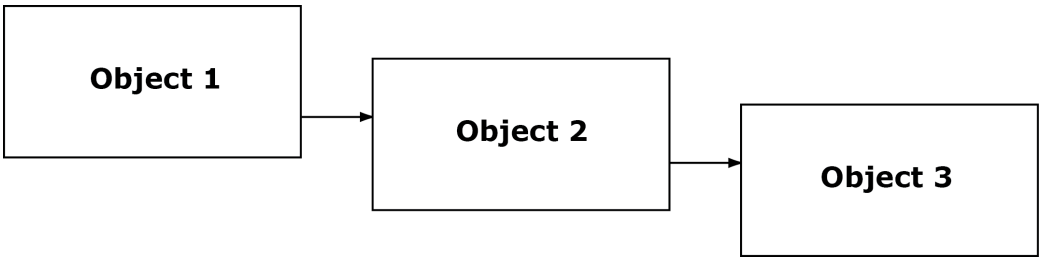
The first step is designing an interface for the email handler.

```
<?php

abstract class Handler {

    protected $successor;

    public function appointSuccessor(
     Handler $successor) {
        $this->successor = $successor;
    }

    protected function passMessage(
```

```
        EmailMessage $message) {
            if($successor instanceof Handler) {
                return $this->
                       successor->
                       handleMessage($message);
            }
        }

        public function handleMessage(
         EmailMessage $message);

}
```

Note that the example isn't using the interface mechanism here; it's using an abstract class instead. Why? Two reasons. First, an abstract class does in fact define a public interface that all subclasses can adopt. Second, there are certain behaviors (like appointing a successor) that are crucial to the Handler. It makes sense to define these as part of the Handler right from the beginning, rather than to implement them in each and every class.

Now that we have a handler in place, we can choose to handle messages in any order we want. For example, we can create a handler that routes certain messages to certain email boxes or rejects certain messages as spam. The most important elements are that you construct the chain and appoint successors along the way.




## Dropping the ball

Of course, with this particular pattern it's entirely possible to develop an application that has a chain that is incapable of handling each and every request type. As a result, the request can potentially "fall off" the end of the chain, leaving the Client with nothing but a null response. We obviously want to avoid returning an unexpected null response to the Client, which is expecting us to handle the request in some concrete way. There are several different ways to avoid this problem.

The first potential solution is to instruct the Concrete Handler on what to do in the event that it cannot pass the request along. In this case, raising an exception is a perfectly appropriate response; there has, in fact, been an exception if a request has reached the end of the chain without having a successor object for handling the request.

Another option is to create a default Concrete Handler and always attach it to the end of the chain. In cases like routing HTTP requests, having a default 404 handler helps ensure that the chain can always be handled, and that the end of the chain is the 404 Not Found operation.

It's important in either case that a request not fall off the end of the chain unhandled unless that's our intent. The Client expects to get a response, and if that response is null, this can create problems in the rest of the application. This is a case where properly implementing and expecting exceptions isn't optional; it's required.

In this particular email handler, we want the message to fall off the end, and if it does, we assume the filters didn't match and that the message is routed to the inbox. The Client would be taught to assume that a null response means a message doesn't match any of the filters. In this condition, the Client continues to process the message based on the default rules it already knows and routes the message appropriately.

## Other uses for the Chain of Responsibility

Routing isn't our only potential use case for this particular pattern. It's useful in many different situations.

The Chain of Responsibility could be used as a filtering mechanism for blog comments. For each comment, the Client can assemble a string of spam rules. If any of the rules are triggered, the comment is treated as spam. Just as in the email filter example above, having the comment fall off the end of the chain unhandled (and return a null response to the

Client) is our desired effect. This means the comment wasn't considered spam and can safely be published.

The Chain of Responsibility can also be used for authentication and permissions. Creating an access control list with the Chain of Responsibility lets us try access controls. If the request falls off the end or is handled by the default handler, the user doesn't have the permissions needed for whatever operation they were attempting. It's important to be cautious when implementing this approach, since the Chain of Responsibility is specifically designed to handle the request with the first possible object in the chain; thus assembling the chain correctly and carefully is paramount.

### THE PRINCIPLE OF LEAST PRIVILEGE

In computer science, the Principle of Least Privilege warns us not to share more than a particular user needs in order to accomplish the task at hand.

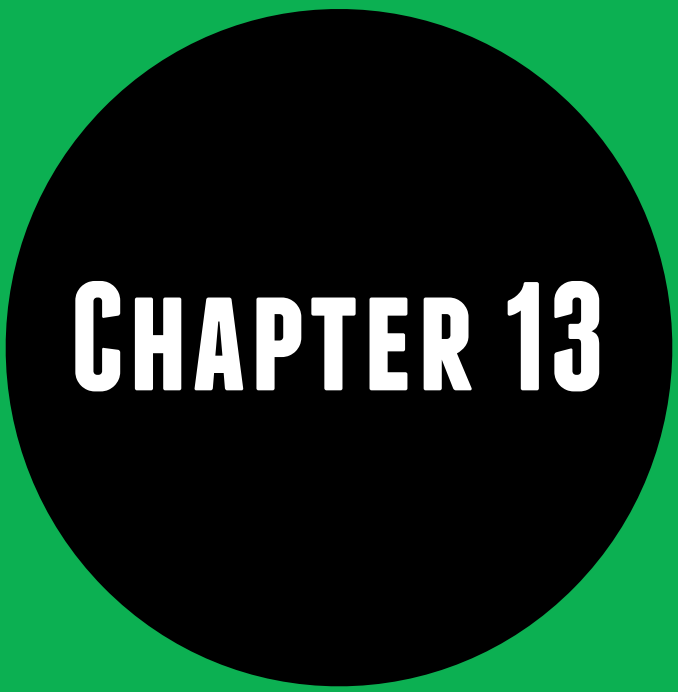
When constructing chains that handle authentication, be careful that those chains conform to the Principle of Least Privilege, and be sure not to build the chain in such a way that exposes too much information.

It's also possible to use the Chain of Responsibility as a discriminatory logging mechanism. By teaching different loggers about the types of messages they should log, the loggers can forward messages they don't care about on to the next link in the chain for handling. For example, perhaps you want to write certain system events to the syslog, but you want

to write other events (user authentication requests, permission elevations) to the database, and still others (the most critical errors) you want sent to your phone. You may also want to limit which messages are written to various logs by severity: your stage server may write all messages, including debug messages to the logs, but your production servers would only write critical messages based on their configuration. By implementing these rules and creating a chain, you can handle each of these requests differently while having only one central place responsible for accepting log messages from the other objects in your application. The syslog will accept system messages and handle those, but pass critical errors and user events to the next object in the chain. Messages that you don't care about won't end up being handled and will fall off the end of the chain.

As you implement this pattern, you'll discover how flexible and easy it is to use. The Chain of Responsibility allows us to add, remove, change or edit the way in which our application treats certain cases, all through configuration. We have the option and flexibility to create new rules and eliminate old or outdated ones. This is possible entirely with configuration and requires no code changes at all!

# YOU CAN SAY THAT AGAIN



# YOU CAN SAY THAT AGAIN

There are many different ways to represent collections. You probably picked up on one of the simpler data structures early on in your PHP career: the array. Arrays are one of the most common structure for representing a collection and iterating through it in PHP. It's one of the first data structures that PHP developers are taught, and it's one of the most powerful data structures available for procedural programming in PHP.

But when we move into the object-oriented world, we want to consider the possibility of collecting things inside objects. And so, over time, developers have come up with various plans and strategies for doing just that. Developed realized that they needed to come up with some kind of way to iterate through objects, and to accomplish this they created something called the Iterator Pattern.

In PHP, The Iterator Pattern is special, because the Iterator interface is built directly into PHP. It's one of the only design patterns that has a built-in interface within PHP, and it's part of what's known as the Standard PHP Library. There are some

key components that are important to grasp about this particularly powerful pattern, so let's get right down to it.

## Iterator, Traversable, and IteratorAggregate, oh my!

The Standard PHP Library (SPL) offers a collection of interfaces along with classes which can accomplish some common tasks for PHP developers. This powerful library is also very difficult to understand and documentation has always been sparse. However, once you grasp this library, the power of the SPL will help you accomplish a number of very common tasks.

The SPL contains interfaces for a variety of iterators. At the root of these interfaces is the Iterator interface, which defines how an iterator will look. The Iterator interface looks like this:

```
<?php

interface Iterator extends Traversable {

    public function current();

    public function key();

    public function next();

    public function rewind();

    public function valid();

}
```

Note that the Iterator interface implements another interface called Traversable. This is a PHP built-in that can only be implemented by other PHP built-ins, and tells the interpreter that a particular object can be used within a foreach loop. All iterators will implement the Traversable interface, and thus can all be used within the foreach loop. This is a powerful advantage.

The Iterator interface is very useful for implementing a particular method for looping through a collection. The way in which you organize, structure and implement the collection is entirely dependent on your specific needs; it is unrelated to the Iterator itself. The Iterator you produce can be used to iterate through the collection in a predictable way.

## Out-of-the-box behaviors to love

Of course, the Iterator is a useful pattern precisely because of its flexibility. The designers of PHP recognized that there were a number of uses for the Iterator interface and included additional interfaces, which add new methods to the Iterator interface.

For example, PHP's designers knew that recursive iteration would be an issue that many developers would face, and created a RecursiveIterator interface:

```php
<?php

interface RecursiveIterator extends Iterator {

    public function getChildren();

    public function hasChildren();

}
```

This iterator allows for the retrieval of children, and expects that those children will be wrapped in a RecursiveIterator instance.

Developers also foresaw the need to be able to search through an iterator for a specific value, much like searching an array for a particular key. Thus, they created the SeekableIterator interface:

```php
<?php

interface SeekableIterator extends Iterator {

    public function seek($position);
}
```

This particular interface is designed to allow the developer to seek a particular value based on its position in the iterator.
Of course, none of these interfaces are useful unless we actually implement an iterator. Let's create an example.

## Implementing our own iterator

Let's re-implement the for loop as a PHP iterator.

While we would typically use a foreach loop to iterate through the array, we can also use a for loop that would look like this:

```
<?php

$a = array('1', '2', '3', '4', '5');

for($i = 0; $i < count($a); $i++) {
    print $a[$i] . '<br />';
}
```

The for loop has four steps: to set the value of the variable, to check that the condition is still true, to run the loop, and then to iterate the value. We can implement this as an iterator with relative ease.

```
<?php

class ForIterator implements Iterator {

    protected $values = array();
    protected $position = 0;

    public function __construct(
     array $values = array()) {
        $this->values = $values;
    }

    public function current() {
        return $this->values[$this->position];
```

```
    }

    public function next() {
        $this->position++;
        if(!$this->valid()) {
            return;
        }

        return $this->values[$this->position];
    }

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

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

    public function valid() {
        if(isset(
              $this->values[$this->position])
        ) {
            return true;
        }
    }
}


$i = new ForIterator(array(
     '1', '2', '3', '4', '5')
);

while($i->valid()) {
    print $i->current() . '<br />';
    $i->next();
}
```

Of course, this looks very much like the built-in ArrayIterator, which accomplishes this task for us. We would never want to write this ourselves if a built-in iterator already exists, which is one of the powerful things about the SPL: someone has already solved many of our most basic problems.

## The most useful built-in iterators

There are a few really useful built-in iterators. Each of these built-ins is designed to offer a common solution to a common problem. In that way, they are much like design patterns, but in this case they are specific implementations of the same design pattern to solve common problems within that particular scope. Here are a few examples.

We talked a little bit about the ArrayIterator and its close cousin: the RecursiveArrayIterator. These two iterators can take an array and iterate through it. And we can use another built-in, the RecursiveIteratorIterator, to iterate through each of the multidimensional levels of the array.

```
<?php

$a = ['1', '2', '3', '4' => ['a', 'b', 'c'],
'5'];

$rai = new RecursiveIteratorIterator(new
RecursiveArrayIterator($a));

while($rai->valid()) {
```

```
    print $rai->current() . ' - foo<br />';
    $rai->next();
}
```

So how does this work? The RecursiveIteratorIterator is an iterator for iterators. It works by iterating through the various iterators that you provide. Being recursive, it asks each iterator it comes across if there are children, and if there are, it handles each one of them. This makes it possible to iterate through each of the levels in the array.

There are a number of other useful iterators, including FilterIterator, RegexIterator, DirectoryIterator, and FilesystemIterator. These iterators can be used to filter data with a set of rules or a regular expression, execute file system operations, or even create ASCII tree art (as with the RecursiveTreeIterator).

## Iterators are not implementation logic

One of the biggest misconceptions with iterators is the assumption that they are part of the logic of the thing we are writing. They aren't. Iterators are different from the logical part of an application. Instead of including the business logic of the collection within the iterator, the iterator is only considered a way of looping through a collection. The definition of the collection, including its interface and implementation details, is out of scope for the definition of the iterator.

Implementing the collection logic along with the iterator violates the Single Responsibility Principle and will produce potentially disastrous results. Iteration is a big responsibility, and adding superfluous secondary responsibilities will create some very serious problems with extension and inheritance.

Note that even in the simple ForIterator we created, the structure of the collection (the array) was untouched. The internals of the collection were unimportant. We could have replaced the array with a group of objects or other iterators or whatever we wanted. As long as we have a collection, we can create an iterator for it.

So what about collections? Simple collections are easy: they can often be represented as arrays. But what about more complex collections, like hierarchical collections? It just so happens there's a pattern just for them.

# TREES EVERYWHERE

## Chapter 14

# TREES EVERYWHERE

One of the most frustrating parts of creating hierarchies is developing the architecture of that hierarchy. Representing objects in a tree is conceptually easy, but representing them in code is programmatically challenging. Yet creating hierarchies and trees is one of the most common programming tasks anyone will have to accomplish. Everything from menus to filters to data validation to displaying phone trees relies on hierarchical data that contains nodes, branches and endpoints.

Perhaps this is one of the reasons that developers have come up with a solution for creating hierarchical objects and design patterns. And though its name doesn't indicate that it relates to hierarchical or tree-like structures, the Composite Pattern offers us a way to create a tree.

The Composite Pattern is closely related to the Iterator Pattern, because many trees must be iterated through in order to find a particular node or to traverse the tree. In fact, it's possible to implement the Composite Pattern alongside the Iterator

Pattern so that you can iterate on the hierarchy natively. The way in which the iteration takes place depends on how the iterator is actually implemented.

## Defining a tree

Before we get into iterating through a tree, let's define what the tree is going to look like. To do this, we will want to define some kind of common interface that clients can use. We're going to create a small application that can be read either breadth-first or depth-first, and the tree will have a node that is considered to be the "it" node; that node will be the node that we are searching for in the tree.

```
<?php

interface TreeComponent {

    public function add(
          TreeComponent $component);

    public function remove(
          TreeComponent $component);

    public function getChild($indexKey);

    public function getChildren();

}
```

Our interface is pretty simple, allowing us to add and remove components as well as to get a specific child or to get all the

children. In a tree, any node can be a child and any node can be a parent. Thus, all concrete nodes will implement the TreeComponent interface. Some trees are traversable in both directions (from the top to the bottom and back up to the top again), but this tree is traversable only from the top node down. By implement a getParent() method in your own composites, you can make it possible to traverse up the tree again all the way to the top node.

```
<?php

class TreeNode implements TreeComponent {

    protected $nodes = array();
    protected $place = 0;

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

    public function getPlace() {
        return $this->place;
    }

    public function add(
     TreeComponent $component) {
        $this->nodes[] = $component;
    }

    public function remove(
     TreeComponent $component) {
        foreach ($this->nodes as $key =>
$node) {
            if($node === $component) {
```

```
                    unset($this->nodes[$key]);
                }
            }
        }

        public function getChild($indexKey) {
            if(isset($this->nodes[$indexKey])) {
                return $this->nodes[$indexKey];
            }

            throw new TreeException(
             'Invalid child requested');
        }

        public function getChildren() {
            return $this->nodes;
        }
}
```

## The Triple Equals

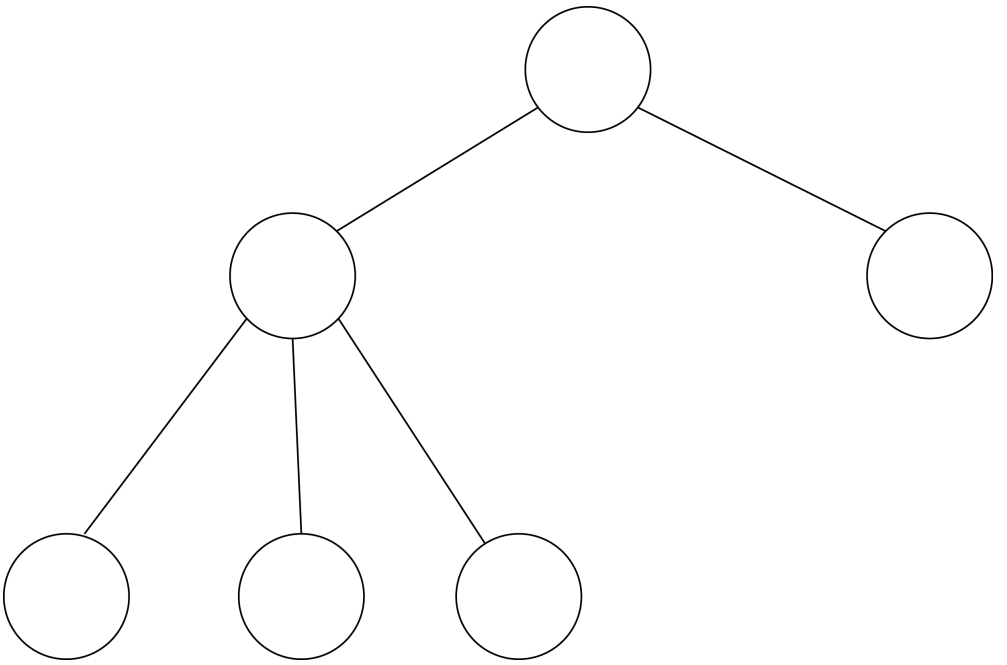
In the TreeNode::remove() method, we use the triple equals (===) when evaluating the node and the provided object.

In PHP, most of the time a double equals (==) is enough. However, with objects, this is a bit more tricky. The double equals will evaluate that the object's type and values match. Two objects that match in type and values will be considered equal.

The triple equals allows us to evaluate more than that: we also check that the objects are the same *instance*: that is, they are the same object, assigned to different variables. In this case, instance, not merely type and values, is what we care about.

Without the triple equals evaluation, we couldn't be sure the right node was removed.

This TreeNode object gives us everything we need to properly and completely construct a tree. We can create objects that are part of a tree any time we want, at any depth we want. For example, imagine that we want to create a tree that looks like this:



We can easily create this tree within our current structure:

```
<?php

$one = new TreeNode(1);

$two = new TreeNode(2);
$three = new TreeNode(3);

$one->add($two);
$one->add($three);
```

```
$two->add(new TreeNode(4));
$two->add(new TreeNode(5));
$two->add(new TreeNode(6));
```

Our tree is now established, and we can traverse the tree by retrieving the children held by each node and seeking the object that we're looking for.

This would be easier if we could iterate over the tree. This is where the Iterator Pattern comes in handy. The Iterator Pattern is often related to the Composite Pattern, because implementing a Composite often results in a need to iterate. So let's talk about iteration.

## Adding an iterator to our composite

At first, it might seem obvious that we could simply add the RecursiveIterator interface to the mix and mix in the iterator directly with the tree, like this:

```
<?php

interface TreeComponent
     extends RecursiveIterator {

    public function add(
     TreeComponent $component);

    public function remove(
     TreeComponent $component);

    public function getChild($indexKey);
```

```
    public function getChildren();
}
```

Though this might appear to be the easiest way to implement an iterator alongside our composite, it's actually a common mistake.

The definition of the tree and the iteration of the tree are separate responsibilities; these are distinct patterns that should be implemented individually. We can create an iterator that is responsible for iterating over the tree and working with the tree to properly iterate through the children. Such an iterator looks like this:

```
<?php

class TreeIterator implements
RecursiveIterator {

    protected $nodes = array();

    public function __construct($nodes) {
        if(is_array($nodes)) {
            $this->nodes = $nodes;
        } elseif ($nodes instanceof TreeNOde)
{
            $this->nodes = [$nodes];
        } else {
            throw new
InvalidArgumentException;
        }
```

```
    }

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

    public function next() {
        return next($this->nodes);
    }

    public function rewind() {
        reset($this->nodes);
    }

    public function valid() {
        return current($this->nodes) !==
false;
    }

    public function getChildren() {
        return new TreeIterator(
         $this->current()->getChildren());
    }

    public function hasChildren() {
        return (bool)count(
         $this->current()->getChildren());
    }

    public function current() {
        return current($this->nodes);
    }
}
```

This iterator can be combined with the RecursiveIteratorIterator to correctly and completely iterate through each node in the tree and conduct operations as

required (for example, to identify the deepest node or to identify a particular node that might have a different subtype than the rest). Whatever the intent, the goal is the same: to create and iterate through the tree.

```
<?php

$it = new TreeIterator($one);

$rii = new RecursiveIteratorIterator($it,
RecursiveIteratorIterator::SELF_FIRST);

foreach ($rii as $node) {
echo ' PLACE=' . $node->getPlace();
}
```

This code will output the nodes in the order that they are discovered (this is a depth-first search). The RecursiveIteratorIterator::SELF_FIRST argument tells the iterator to start with the first iterator (otherwise it would skip the first node).

## Finding the “it” node

We can use this new set of tools to do a number of useful things: now we can find the “it” node, based on a particular object type, or we can identify the maximum depth of the tree. To find the “it” node we must first define an “it” node object, like so:

```
<?php
```

```
class ItTreeNode extends TreeNode {}
```

Next, we can add this to the code that defines our tree.

```
<?php

$one = new TreeNode(1);

$two = new TreeNode(2);
$three = new TreeNode(3);

$one->add($two);
$one->add($three);

$two->add(new TreeNode(4));
$two->add(new TreeNode(5));
$two->add(new ItTreeNode(6));
```

Of course, we'll have to change the foreach loop slightly to allow for us to find and identify the "it" node, and we will want to know the depth of the node in the tree when we do it:

```
<?php

$it = new TreeIterator($one);

$rii = new RecursiveIteratorIterator($it,
RecursiveIteratorIterator::SELF_FIRST);

foreach ($rii as $node) {
    var_dump($node);
    if($node instanceof ItTreeNode) {
        echo 'Found it! Depth was '. $rii-
>getDepth();
```

```
            break;
        }
}
```

This will properly identify the depth of the node. Through iteration, we can find the node that we are looking for. This example illustrates the tremendous power, flexibility and simplicity of combining these two design patterns. Much of the iteration and composite-building that you will do involves seeking particular nodes and iterating to find them. These two patterns make this an extremely easy task to accomplish.

## Building menus and hierarchies

The Composite Pattern is most useful when you want to create menus or hierarchies.

Many examples of the Composite Pattern in action describe creating a menu. This is because the Composite Pattern is so well suited for this particular task. Menus are typically hierarchical, and it's important to be able to represent that structure in some way, shape or form.

Another common example is the example of a phone tree, from a CEO down to the lowest-level employees. This is a great example, because there's a single person at the "top" and many people at the "bottom," all of whom (should) only have one boss. Meanwhile, bosses may have multiple direct reports,

and they also have a single boss, all the way on up to the CEO herself.

## Do all composites look the same?

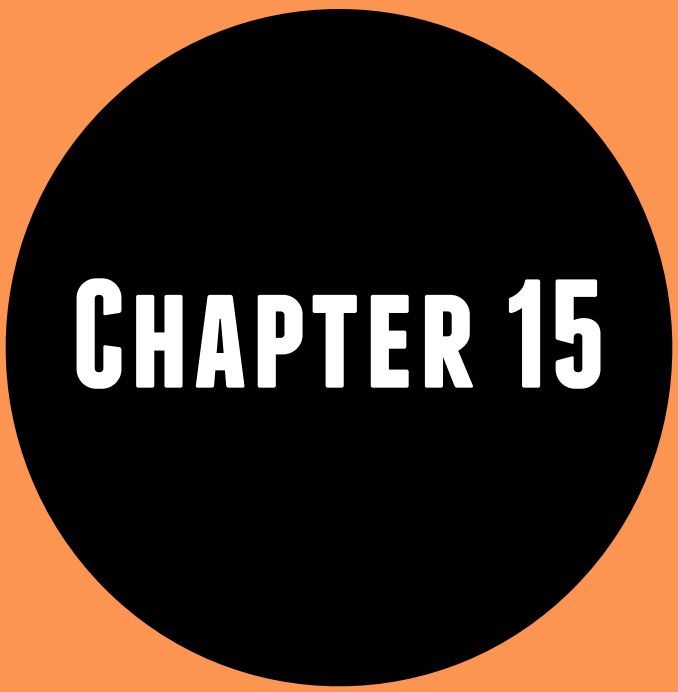
Not at all. Composites are designed to suit the problem you are solving, not to fit some predefined interface.

The interface used in this chapter is meant to illustrate a common look, but isn't the only way to implement the pattern. And even though we've talked about many different ways to implement a design pattern before, it bears repeating: even if you find that it's easiest to implement this design pattern almost exactly as I have done, you are not required to do so. In fact, you can implement it in any way, with any method names that you want.
Of course, the caveat to that is that you must use the built-in Iterator interface if you want to use the built-in loop methods of PHP (see "You can say that again..." for more on the Iterator Pattern).

Still, even considering the common (and set) methods for iterators, you can let your creativity flow when it comes to the Composite Pattern. Maybe you want to be able to determine the parent of a particular object, or you want to be able to prohibit the addition of children past a certain point. Whatever

you do, be creative and remember that solving your problem is the most important task you have.

# THE PATTERN EVERYONE ALREADY USES



# THE PATTERN EVERYONE ALREADY USES

As PHP developers, there's one design pattern (or really, one architectural pattern) that we see most often. It's in every major framework and it forms the basis of how we create our applications. Above all other patterns, this one seems to have won the hearts and minds of developers everywhere. It's so ingrained in us, in fact, that we even name our objects after it, as though there's no other name by which to call them.

What is this magical pattern that everybody loves so much? It's called Model-View-Controller.

Model-View-Controller (MVC) is less a design pattern than an architectural pattern, informing how we go about building applications and forming the object architecture. MVC exists in almost every major framework in some form or another. The reason that this pattern is so popular is because this pattern works really well for us on the Web. We know that we want to separate HTML from business logic, and MVC lets us do this by putting our HTML in the template and our business logic in the model. The Controller acts as a mediator between the two,

facilitating traffic flow and doing a little bit of data validation on the side.

This pattern is, in fact, so common it's often overlooked as a pattern at all, but it bears examination and a discussion of the right, and wrong ways to create MVC applications.

## Understanding all the moving parts

In order to fully understand this architectural pattern, we need to examine each of its parts and identify its responsibilities. Each component has distinct responsibilities that you'll want to apply consistently. The biggest mistake many developers make with MVC is assigning responsibilities to one component that really belong to another. We want to avoid doing that in our implementations of MVC.

Let's start with the Model. The Model is the part of the application that represents data entities to the rest of the objects we create. The Model is also responsible for persisting data between requests; it is the component interfaces with some kind of data storage system. The Model is often (but not always) the largest amount of code in an MVC application.

For PHP developers, the Model contains all of the code that we want to execute for business logic, validation checks and storage. The Model is so important that there's a dedicated chapter on models (Beautiful Models) which dives into model

development in detail. For now, it's important to realize that the Model is the heaviest lifter of the MVC pattern.

Next, we have the View. The View (also sometimes called the Template, in Model-Template-Controller) is responsible for displaying the application to the user. Here is where all of the display logic resides. The View is also the point at which the user can interact with the system through forms or buttons and other methods of control.

The word "template" is sometimes used to describe this layer because the View isn't a true View in the Martin Fowler classical sense. Instead, it's a collection of HTML, Javascript, CSS, and other components that make up a webpage. The View does very little work in terms of data manipulation, usually only having enough functionality to display data in correct formats to the user.

In an MVC application, the View is the only place that HTML would be considered acceptable. It's also the only place where data formatting for display purposes would generally be expected. Models deal with data as it's presented to them, while the View is concerned with how that data looks to the end user.

There's still one part left: the Controller. There's a good reason that the Controller is mentioned last - it does the least amount

of work. In fact, the Controller is nothing more than a traffic cop for the MVC application.

Think back to the Mediator Pattern we discussed, and recall that the job of the Mediator is to facilitate communication between colleagues. This is essentially the same role as the Controller: to facilitate communication between the Model and the View. The MVC pattern is an implementation of the Mediator Pattern, but it's a special implementation with specialized roles. This is what makes it a unique pattern.

But back to the Controller. The Controller's job is simple and straightforward: handle the data (usually requested via HTTP) and send correct requests to the Model. Once the Model returns a response, it collects that response and sends it along to the View for display to the user. The Controller isn't focused on validation or business logic and doesn't handle display behaviors. It's only interested in moving data from one place to another.

## Controllers are not models

There are a number of ways to potentially mess up the MVC architecture, most of which can be avoided by exercising a little extra care.

The easiest mistake in MVC is to put business and validation logic into the Controller. It feels right when we're doing it, but it

creates serious challenges down the road that we want to avoid.

For example, most modern Web applications make use of an API that returns some kind of machine-readable response (XML, JSON, etc.). When creating an API, often there are two controllers in play: the one that returns machine-readable responses and the one that passes data off to the view for display.

If we place our business logic into the controller that interfaces with the view, we run into a challenge when we need to build an API. We have two choices: a painful refactor to what we should have done from the beginning or duplication of code. The former takes a tremendous amount of time and energy, while the latter makes the code unmaintainable in any meaningful way.

Placing business logic into the Controller also makes the Controller difficult, if not impossible, to test. The Controller often has a number of dependencies: request and response contexts, view objects, and connections to other resources. just to name a few. These dependencies must be mocked or created for testing.

The Model, on the other hand, has far fewer of these dependencies, and the few dependencies it does have can be

easily mocked for testing. This is why the Model makes sense as a place for the majority of our business and validation logic: it's far easier to test.

## Models are not views

Another common mistake is the inclusion of display-related behaviors in the Model. This is not uncommon with data (like phone numbers) that has to be formatted for display. It might seem convenient to invoke rules designed to properly format the phone number for display. Even though it seems reasonable to place these methods in the Model, this approach creates problems down the road.

Display logic belongs in the View. This includes formatting of data for a particular style and manipulating data into tables or organized structures. The Model doesn't know about (and shouldn't care about) how the data is displayed, because that's out of scope for the Model.

Even though we want to place our formatting into the View, it's important to be careful not to duplicate code there, either. This is where the concept of View Helpers comes into play: these are useful objects of functions that can be invoked by the View for the purposes of making display formatting consistent across all Views. This has the added advantage of creating a single bit of code to maintain, reducing the cost of making changes to the format later on. Most modern View layers offer

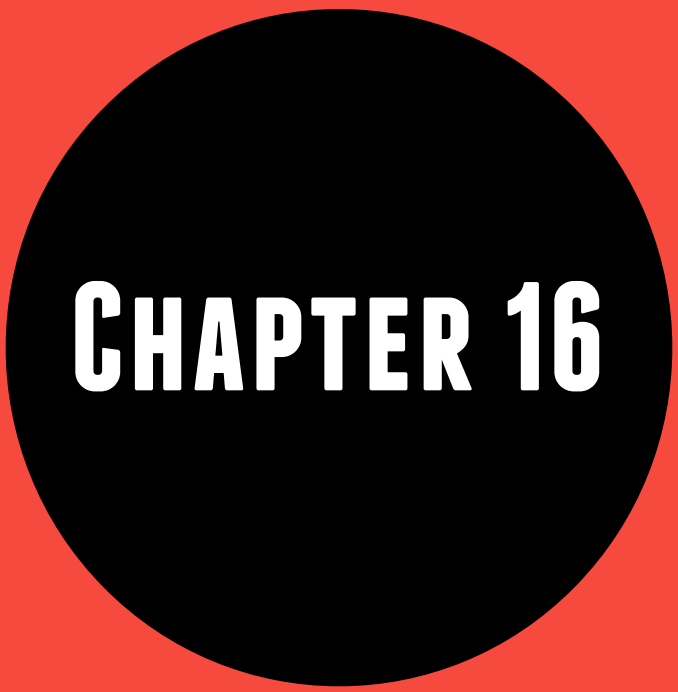
some kind of View Helper plugin system, making it easy to write these plugins yourself or to use one of the included plugins from the framework author.

## Not just for frameworks

We've spent a good amount of time talking about MVC in a framework context, and it might cause you to think that MVC only exists in frameworks. If you choose not to use a framework you might think you can't use MVC, but that's not true at all.

Anybody can create an MVC application, framework or not. While frameworks have widely adopted this pattern and enforce it strongly, that's not because it's a framework-specific pattern; it's because for frameworks it makes the most sense. MVC can be as simple or as complex as you want it to be for your application. Don't feel like you have to adopt a framework in order to use it. Just create a Model, a View, and a Controller, and you're in business! The power of MVC is not in the frameworks that use it; it's in its flexibility and its ability to create powerful apps that are easy to maintain and test.

# BEAUTIFUL MODELS



# BEAUTIFUL MODELS

Despite its name, this is not a chapter full of photos of beautiful people. Instead, it's about creating beautiful, effective, well-designed object models that represent data in the backend of applications.

Every web developer has the same problem: the web is stateless. That is to say, the lifecycle of a web request doesn't last that long, and that once a specific request is completed, it shares nothing with the requests that follow it. In fact, this statelessness is a cornerstone of the modern Web. But it has always presented developers with a problem: how can they persist certain data beyond a single request?

Of course, sessions, cookies, browser fingerprints, and other practices identify a specific user between requests. In this chapter, however, I focus on how we store — or, more precisely, how we represent the data that we store — between requests in the context of a web application.

## The Model-View-Controller

Most applications utilize a structure known as Model-View-Controller. A design pattern in its own right, this particular pattern utilizes a controller as a mediator between the Model, which stores and persists data, and the View, which displays data to the user. Today, most PHP developers use a cousin pattern called Model-Template-Controller, largely due to the fact that modern frameworks often include a templating layer rather than a view layer. Still, whether it's Model-View-Controller or Model-Template-Controller, the idea is the same.

The problem is that developers often assume that they should put their business, validation, data storage and data retrieval in the Controller. Bzzt. Wrong. As we saw in "An Object Between Friends," the Mediator's job isn't to handle business or validation logic; it's to handle the passing of messages between two layers of the application. In this case, those layers are the model and the view.

As a result of the focus on the Controller as a mediator, a saying has become popular in the MVC world: "fat model, skinny controller." The saying assumes that the Controller is nothing more than a traffic cop between other layers, which is largely true. And the View, being a colleague of the Model (but not knowing it directly), can't be responsible for persisting data either, or representing it beyond simple display and formatting operations.

This leaves the responsibility of representing and storing data to the Model, which in turn has its own design pattern. This is called the Domain Model Pattern.

## Compound patterns

This is a good time to address the concept of compound patterns. A compound pattern is a pattern that applies one or more existing patterns in new ways to form a distinct pattern of its own. Model-View-Controller is a great example: the Mediator Pattern is used to mediate between the layers, but the layers are organized in distinct ways that make MVC a design pattern in its own right.

Similarly, the Model layer, when using the Domain Model Pattern, uses the Mediator Pattern to mediate between two distinct layers. The first layer is the Value Object. The Value Object represents the data that is persisted between requests. But it does not handle storing or retrieving the data from some kind of data store; this falls to the Storage Object. The Gateway Object, which is the Mediator here, handles communicating between these colleagues. In this way, the Domain Model Pattern is a compound pattern, implementing one design pattern to create a second design pattern.

## The Domain Model Pattern in a nutshell

Let's expand the definition we've been working on here. At its most basic, the Domain Model Pattern contains a minimum of

three elements: the Gateway, which is a mediator object; the Value Object, which represents and models the data; and the Storage Object, which handles storing and retrieving data from some data storage system.

Even though most examples contain only three objects, the actual objects involved in the pattern can represent a variety of different shapes and sizes and numbers. For example, proper abstraction of the Storage Object may require a separate object for different tables or different data storage engines. There may be different Value Objects that are required in the model, and each of these may have a separate role. The same Gateway may interact with multiple Value Objects and multiple Storage Objects all at the same time.

The purpose of breaking the model and the logic away from the Controller is to abstract this particular subsystem away from the Controller-level functionality. And the Gateway, which is a mediator, is responsible for mediating between each of these objects.

## Business and validation logic

As a model is developed, we need to add business and validation logic. These components don't belong in the Controller; they belong in the model. But where?

The answer is that business and validation logic belong in the Value Object. As far as the application is concerned (as well as the other objects in the application), the Value Object represents an entity within your application. Whether that entity is a bug, a customer, a transaction, or something else, all of the business and validation logic for manipulating the data in that object should lie within that object. No exceptions.

It's a common mistake to place these elements in the Gateway. However, the Gateway may not always be available to whatever object is working with a particular Value Object; thus, it should be assumed that the Value Object contains all the logic required to handle the responsibilities it's given.

### How is this not a facade?

Recall back to the chapter, "Translating Between Objects", and you'll recall that subsystems are often given a Facade interface, for higher level objects. It can seem that the Controller here would be a higher-level objects that wants to interface with a lower-level subsystem, so what makes the Gateway object a mediator, instead of a facade?

The difference is simple: the Controller is considered to be a colleague alongside the Value Object and the Data Storage Object. In fact, the Controller may interact with Value Objects from time to time, while a Facade would never expose a subsystem object directly .

These differences make the Gateway a Mediator, not a Facade.

It's also important to understand that the Value Object is recognized by other parts of the application, since it represents the entity for which the Value Object was created.

As a result, type hinting on a specific concrete Value Object is not considered a violation of the Liskov Substitution Principle or the Dependency Inversion Principle; there is no point in creating an interface for most Value Objects, because there is no other way to represent these particular entities.

## The process of creating a model

In order to properly use the Domain Model Pattern, it's important to consider what your intentions are and what you plan to model. Models are more than direct relationships between database tables. In fact, many models don't have a one-to-one mapping between the Value Object and the Storage Object (which is why the Gateway exists). When developing models, it's crucial that developers dislodge the perception that a one-to-one relationship exists between models and tables.

Instead, the first question that anyone should ask when creating a model is, "What am I modeling?" For example, if you're writing banking software, you probably recognize that customers have accounts, that more than one customer can be associated with an account, and that a single customer may have more than one account as well (a checking and a savings account, for example).

Customers also have all kinds of personal details: email addresses, phone numbers, and home and business

addresses. These items have a one-to-many relationship between customer and data point. In a model with a one-to-one relationship between model and table, you'd have separate models for email addresses, phone numbers, etc. But in a Domain Model, each of these data points belongs to the Customer model, because the Customer owns these various data points. They are part of the Customer entity.

This creates a situation where the Customer Value Object and the Customer Storage Object are different. The Customer Value Object will contain all the related data points (likely represented as arrays), and the Storage Object will have to know how to retrieve and store these points in their respective tables.

Similarly, the Account Value Object will know about the Customer, but the Customer entity is separate. In this way, we are actually creating two models: the Customer model and the Account model, despite the fact that customers and accounts have various other data points associated with them that may be in their own tables.

## A rejection of Active Record

Active Record is a pattern that encapsulates the database logic and the value logic in the same object, which maps directly to the database tables. It's a common pattern found in many frameworks, including the popular Ruby on Rails and

Django. Martin Fowler described the Active Record Pattern in *Patterns of Enterprise Application Architecture* (2002), and it's very common to see this pattern in various applications.

Even though Active Record can be used to easily, consistently and effectively create models quickly (and to develop ORM-like systems), there are a number of concerns that make this pattern unsuitable for good object-oriented applications.

We want to do what we can to follow the SOLID principles, but Active Record makes this very challenging. For example, the encapsulation of storage logic and data representation in the same class gives that class two responsibilities — a clear violation of the Single Responsibility Principle. There's also no easy way to create a defined interface that can be used in accordance with the Dependency Inversion Principle. Active Record is dependent on the database, making unit testing difficult, if not impossible. To unit test, a developer must be able to mock objects and test functionality in isolation, but mocking the database dependency is hard when it's directly integrated into the object being test.

Active Record is still very popular today, for many reasons. The two big ones are that it makes it incredibly easy to create models — and then create databases for those models — along with the fact that it provides a lot of "magic" for free. Prototyping applications with Active Record is simple, and

developers are drawn to it for this reason. Active Record is a great prototyping pattern. But when you're facing a more complex application or considering developing more decoupled models, consider the Domain Model Pattern instead.

## Models are complicated

Creating good models is one of the most complicated tasks any developer tackles. For a long time, the Zend Framework documentation held that there was no Zend_Model class because creating a model is the bulk of an application development process. To create a Zend_Model would be to assume that everyone could or would want to use the same model structure, which would be impossible.

For the same reason I haven't included any code in this chapter. The creation of models is extraordinarily case-specific, and depends on a variety of factors specific to your application. The best we can do is to draw a mental picture to refer to as we move forward in creating our models.

Models are the most complicated part of object-oriented development, and something that each developer gets better at as their experience grows. Even if you struggle with creating models at first, the process and practice of doing so will always outweigh the cost. In fact, creating a great model is one of the most rewarding development experiences you'll have. Creating great models that encapsulate your data and make

your application hum is an experience well worth the challenges you'll face.

# ONE PATTERN TO TIE THEM ALL TOGETHER

# ONE PATTERN TO TIE THEM ALL TOGETHER

Most frameworks use the Model-View-Controller Pattern as their go-to for creating flexible, powerful applications. Yet most frameworks also rely on a smaller, lesser-known pattern that is crucially important to the existence of the framework. Without this pattern, frameworks couldn't exist at all. After all, in order to create a framework, there has to be a starting point: something to get the ball rolling for the entire framework to stand up on.

And what is this glorious pattern? It's called the Front Controller Pattern, and it's the most important pattern to framework development — while at the same time —the most important pattern nobody has ever heard of.

The Front Controller Pattern is responsible for providing the glue that sticks a framework together, manages how the framework operates, handles each and every request, creates all the various dependency objects, and actually invokes the controller. Yet this underappreciated pattern does all of this

work in the background, never being noticed by most developers. It's time to give this important pattern its due.

## The invisible foundation

The Front Controller is unique in many ways. It's the center of every process that framework applications execute. It handles each and every request. And often, it's completely invisible. This crucial foundation on which every behavior rests is out of view of most developers, unnoticed until something goes wrong.

The Front Controller is unique in another way, too. It's the only pattern that's impossible to diagram, and there's no way to create a code sample. The Front Controller is as unique as a snowflake, entirely dependent upon the specific framework that creates and relies upon it.

You might wonder, "If there's no way to diagram it, how can it be a pattern?" It's a pattern in the same way that a building's foundation is a pattern: the foundation of a building is as unique as the building on top of the foundation, yet it's invisible. Foundations are constructed using common techniques and they all have a common purpose: to hold the building up. But the way in which they're constructed is unique, and no two are exactly the same.

The Front Controller is the foundation of the application, often the first object to be instantiated and the last object to be destroyed when runtime ends.

## Many duties, one (collection of) objects

There are many responsibilities for a Front Controller, and many Front Controllers are actually made up of several objects that handle each of those duties and responsibilities.

In a basic Web framework (that is, a framework that accepts HTTP requests and sends HTTP responses), the Front Controller is the first component to be instantiated and invoked. It is then responsible for a whole host of other responsibilities.

First, it has to parse the request and identify various components, like where to route the request and whether the request can be fulfilled.

The Front Controller is also responsible for building other library components, from connections to a database to validation mechanisms and routing behaviors. Most of the time, the Front Controller is also responsible for the autoloading behaviors that take place in an application. The Front Controller also invokes the routing mechanism to determine how the request should be handled.

Once the Front Controller knows how to handle the request, it must then find the right controller for handling that request. No MVC application can exist without a component to find and invoke the correct Controller.

Once the Controller is in the mix, the Front Controller can take a break. While the Controller runs, the Front Controller is listening for a response. Once the Controller is finished, the Front Controller receives the response, packages it up into an HTTP response, and sends it back to the user that requested it.

All of these operations take milliseconds and are almost completely transparent to the developer who is writing controllers. This is by design: the developer doesn't have to care about how the framework stands up, and the Front Controller is entirely abstracted away from the process of developing the rest of the application.

## Developing SOLID Front Controllers

When developing your own front controllers, it's vitally important that you continue to honor the principles of SOLID object development. It might seem easy to create one big "God object" to handle all of these responsibilities, but this is precisely why we develop frameworks: to create libraries that can handle individual parts of the process.

In this way, a true Front Controller is a Mediator, just like its Controller counterpart in MVC. The difference is that while a Controller in MVC handles a specific request for which it has been designed, the Front Controller handles all requests without discriminating between them. The Front Controller also works with a larger set of objects, from routers to autoloaders and caching systems, making it's behavior far more broad than that of the MVC Controller.

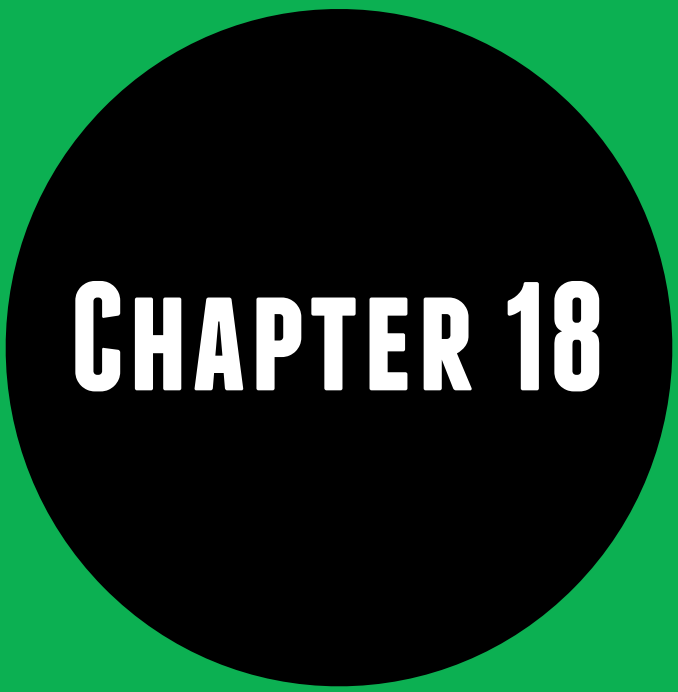
## Beyond the theoretical

This can seem awfully theoretical to many developers, especially if they never want to implement your own framework. Yet in the next few years, you'll likely find that the concept of "micro frameworks" begins to take off. Performance is often based on the amount of code being executed; the less code, the better. As a result, developers will begin moving away from established frameworks, and into creating their own frameworks based on libraries and components they can find.

As a result, you will probably implement your own micro framework, and you'll need a front controller to tie it all together. At that point, understanding the behavior and expectations of a front controller will make a huge difference. Avoiding the common pitfalls of front controller development will help you to create flexible applications with reusable libraries available for your next project. Reusable code is the

point of object-oriented development, and the front controller helps make that happen.

# BEYOND DESIGN PATTERNS



# BEYOND DESIGN PATTERNS

When starting out with design patterns, it's often very easy to see them as solutions set in stone, immovable, almost Biblical in their authority. This viewpoint often comes from a lack of experience with design patterns, and many developers assume that if someone took the time to write about a particular pattern than what they had to say must be the absolute truth without variation.

But this couldn't be further from reality.

Design patterns are not intended to be concrete solutions that can't ever be changed. Far from it! Design patterns are instead meant to serve as forms that you can examine, disassemble, reinvent and recreate over and over again, each time putting your own spin on them as you gain experience.

And even though this book has covered seventeen different design patterns, the fact of the matter is that there are dozens more waiting to be discovered, created, codified and

explained. Design patterns are very much an open subject, ripe for study, review, consideration and discussion.

You may find in your career that some design patterns you love and use frequently, while others you see as too challenging or too rigid for your needs. Design pattern selection is often personal preference on the part of the person using them. And in your career you may even invent your own design pattern that solves your unique problem. It's not out of the question!

But all the patterns that you learn, or that are described here, or that are defined in the books throughout the world, can never take away your creativity as a software developer and an engineer. It's up to you to write code, and up to you to decide which patterns you want to use.

Here are some final thoughts of things you should consider as you go forward in your career to use design patterns.

## Apply patterns to your problems, not the other way around

It can be easy to think, "I have this tool, what can I fix with it?" But thinking this way leads us to shoehorning our products into the patterns we have, rather than solving the problems we have, even if it means thinking up a new pattern.

When confronting a new project, ask yourself how you want to go about solving the problems, and then find patterns that do so. Never be afraid to try a new pattern, or invent a new solution, because that may be what is required. The design patterns you know may not be the best ones for the problem at hand, and you may have to be a bit creative. Don't worry! With design patterns there is no "one true path" from which to stray; focus on solving the problem you have, and the patterns will take care of themselves.

## Recognize that all patterns can be implemented in a million different ways

Ask city planners how many ways there are to lay out a city, and they'll point to the thousands of cities on the map. Each city is unique, with its own set of streets, alleys, homes and businesses, attractions and public spaces, parks and museums. And yet if you take a close look, you can begin to see similarities in the cities of the world. There are patterns to how cities are laid out, designed, zoned and created. Still, despite these patterns, no two cities are alike.

Each application is also unique, and the method for implementing the patterns must also be unique. The individual challenges and problems in each application mean that the solutions must be different as well; it is impossible to simply copy one solution into another project, because each one is different from the next.

In the same vein, just because a developer implements a particular pattern differently from you does not necessarily make their implementation wrong. It just makes it different.

Instead of focusing on the One True Way of implementing a pattern, embrace diversity and learn all you can from unique and creative implementations. Explore them as an opportunity to focus on expanding your own knowledge and seeing an old problem in a creative new way. You may just find it coming back to you in a time of need!

## Accept that sometimes there's no pattern for the problem you're trying to solve

Design patterns do not solve every problem. As common solutions to common problems they can provide us with many types of solutions, but for unique problems and unique challenges, there is likely no established set of patterns.
This is not a cause for concern.

As engineers, it's our job to invent new solutions. It's our job to create answers to challenges that haven't been thought of yet. And it's our responsibility to be innovative and creative.

When you run across a situation that has no obvious design pattern, start thinking about the ways in which you can solve the problem anyway. Design patterns offer solutions that are

adaptable and reusable, so consider in your solution ways in which you can reuse and adapt the solution to future similar challenges. Look for examples of similar problems being solved by others and adapt them if they're available. And above all, enjoy the opportunity to solve a new, previously undiscovered challenge; that's a true rare moment in a developer's career.

## Be creative, and create patterns that work for you

When confronted with a challenge, accept it and create new patterns. Innovate. Invent. Find new ways to solve old problems and find new ways to solve new problems, too. Write about the patterns you develop. Share with the world. Someone else probably has the same problem; tell them how you solved it. Don't be afraid to share it with the community.

Creating new patterns doesn't mean you don't understand the existing ones or that you're doing something wrong. The people who created the first design patterns weren't working off a script, and you shouldn't either. Instead, the people who wrote down the first design patterns observed the code around them and realized that lots of people solved the same problem in the same way. They decided that the commonalities were too great to keep to themselves. But the first list of design patterns was by no means authoritative or complete. New design patterns are being created all the time for all kinds of

different and unique situations that hadn't been invented or conceived of before.

So go forth, and be creative. Explore, experiment, and engineer. Move beyond the rigidity of design patterns into a world of endless possibilities, where design patterns are as natural as the code you write today.