# 2

# Design Patterns

Now let's move from concrete object-oriented implementations to some more abstract concepts, beginning with design patterns. Design patterns have been all the rage for the past several years, and who has not felt a little unsure of themselves when someone says, "You don't know what the Memento pattern is?" In all honesty, I do not. I pulled it out of the air from Wikipedia (see *http://en.wikipedia.org/wiki/Memento_pattern*).

Although many design-pattern proponents are annoying, understanding and implementing design patterns can provide a lot of benefits, particularly some of the design patterns I present here. You can think of these as base patterns.

A design pattern has absolutely nothing to do with software. It is an architectural process first brought into form by architect Christopher Alexander. It states that patterns of design are inherent in many different instances. In other words, project A and project B, though completely disparate in every way, can still have similarities in design because of similar problems that need to be solved.

In Alexander's book *A Pattern Language* (Oxford University Press, 1977), he describes how people should be able to build their own homes and communities through a series of patterns. These patterns are made available via the implementation of doors, walls, windows, colors, and such to make a community in the likeness of the desires of the people who are living there. Although I have not read the book, it has a number of interesting observations that seem worth considering. But at 1,100 pages, it is larger than the giant book on SQL I have in my library that I have not read either.

But let's move those considerations from the world of architecture and think about them from a software-design perspective. At times, wouldn't a senior architect simply like to tell a junior developer to build a class that can have only one instance of itself, and have that developer be able to know exactly what the architect was talking about and proceed to create that class? Wouldn't it be nice to have a means of communicating these types of concepts without having to describe them in detail?

Well, that is exactly what we are going to do here. I will show you six common design patterns that PHP developers use and will provide some concrete examples, so you, too, can start talking in the secret language of design patterns.

## Singleton

In a mathematical sense, a singleton is a set of numbers that has one, and only one, value. So it is a set whose definition is to have multiple items but forcibly having only one. In programming, the Singleton design pattern limits a defined class to only one instance of itself. In other words, if one context of the application has an instance of the object and another context does too, both are using the same instance. If that is a little confusing, do not worry. We will examine some code to understand it:

```
class Car
{
      public function drive()
      {
            $driver = Driver::getInstance();
      }
}
```

*Continued*

```
class Truck
{
      public function drive()
      {
            $driver = Driver::getInstance();
      }
}

class Driver
{

      protected static $instance;

      protected function __construct(){}

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

This example has a class called Driver, which represents someone who will drive a vehicle. Because the application is acting on behalf of a human (us), the driver, for a particular context, will always be the same. So whether you tell the car to drive or the truck to drive, the driver remains the same.

A more realistic version of the Singleton design pattern is an object that represents a session. Because a session can be started only once, it is a natural fit for a Singleton. A Toolkit connection is another real-world example of a Singleton.

When creating a Singleton class definition, you usually create a constructor that is protected. Doing so ensures that the class itself will be responsible for instantiation. Code that is outside the class cannot instantiate it:

```
class MyClass
{
      protected function __construct() {}
}

$cls = new MyClass();
```

This code produces an error:

```
PHP Fatal error:  Call to protected MyClass::__construct() from invalid
   context in test.php on line 10
```

Some examples show the constructor being defined as private, and I am not sure how I feel about that practice. It disallows child classes from creating a new instance of the class. Also, a private constructor is not available to child classes. I would almost prefer to mark the `getInstance()` method as final so that if an error occurs, it does so when the class definition is loaded rather than at some arbitrary point during runtime.

However, the Singleton design pattern is not without its detractors, who have leveled several charges against it.

The first is that Singletons are difficult to test. The theory is that testing a Singleton will inevitably change the object because Singletons are often used to maintain a global state without residing in the global scope. Therefore, you might not be able to develop a test that is consistent. I call this a half-truth because, yes, if you are changing the state, you might have difficulty creating a repeatable test scenario.

The second charge is that in PHP, Singletons are not true Singletons. Because PHP uses a shared-nothing architecture where HTTP requests do not know about other HTTP requests, multiple instances could exist. Therefore, PHP cannot have true Singletons. This I consider purist nonsense.

The last charge is that Singletons create tight coupling in your classes. This I completely agree with. Dependencies in a class are difficult to test. Consider a class that requires a connection to the Toolkit. If you built a unit test for that class and that class explicitly pulled that connection from a hypothetical `ToolkitAdapter` class, you would not be able to mock the adapter class. Mocking an object is the practice of replacing some of an object's members to provide data in a manner that skips

the actual connection to the external resource. Unit tests are used for testing units of logic, but they should almost never require external resources. Mocking lets you avoid that by pretending an object has connected to the external resource, when in fact, it has had the expected data injected into it. If this concept is a little fuzzy, do not worry about it. We will examine it in much more detail in Chapter 7, which covers unit testing.

So, is a Singleton evil? If you wonder why I ask that question, type "are singletons" into Google to see those search suggestions. My answer is no, but they can get you into trouble if you are not careful.

## Factory

Factories are classes that create instances of objects. Why wouldn't you simply use the new keyword and instantiate them from there? Because that additional logic is often required for certain types of classes to be instantiated, and to use the new keyword, you would have to implement that logic everywhere:

```
class VehicleFactory
{

      /**
       * @param string $type
       * @return AbstractVehicle
       */

      public static function factory($type)
      {
            switch ($type) {
                  case 'car':
                        $car = new Car();
                        $car->setTires(new Tires());
                        $car->setChassis(new CarChassis());
                        return $car;
                        break;
                  case 'truck':
                        $truck = new Truck();
```

*Continued*

```
                        $truck->setTires(new Tires());
                        $truck->setChassis(new CarChassis());
                        return $truck;
                        break;
            }
      }

}

$car = VehicleFactory::factory('car');
```

## Adapter

Before writing this book, I had built several adapter classes. But in digging a little deeper, I found that although my implementations were correct, my understanding of the definition was a bit off. My working definition before this discovery was that an adapter was a class that made a common interface to multiple different classes, depending on the implementation. And this is true. But I missed an important detail. An adapter is a class designed for connecting multiple *incompatible* classes with a common interface. It is a nuance but an important one.

The word "adapter" comes from two Latin words: "ad," meaning *to*, *toward*, or *at*; and "aptō," meaning to *adjust* or *prepare*. Therefore, you are adjusting one toward another. Adapters are most commonly used as a means to provide interfaces to different database adapters. PHP Data Objects (PDO) is an example of the Adapter pattern but in reverse because it uses drivers. You write a driver to fit a unique implementation with a common interface; it is the driver's responsibility to be compliant. An adapter is a common interface that translates functionality for the unique implementation; it is the adapter's task to be compliant.

Following is an example that defines an adapter for authenticating against either the Toolkit or a DB2 connection:

```
interface AuthenticationAdapterInterface
{
      public function authenticate($username, $password);
}
```

*Continued*

```
class ISeriesAuthenticator implements AuthenticationAdapterInterface
{
      public function authenticate($username, $password)
      {
            try {
                  $connection = ToolkitService::getInstance(
                              '*LOCAL',
                              $username,
                              $password
                  );
                  return $connection != false;
            } catch (Exception $e) {
                  return false;
            }
      }
}

class DB2Authenticator  implements AuthenticationAdapterInterface
{
      public function authenticate($username, $password)
      {
            $connection = db2_connect(
                        '*LOCAL',
                        $username,
                        $password
            );
            return $connection != false;
      }
}
```

Here, the example is defining an adapter interface called AuthenticationAdapterInterface, which states that an authentication class must implement the authenticate() method. This action forces the developer to write a class that makes a certain requirement to use one of the multiple disparate authentication mechanisms. That requirement is defined by the interface:

```
$authenticator = new ISeriesAuthenticator();
if ($authenticator->authenticate($username, $password)) {
      // logged in
}
```

Using the Adapter pattern, you do not have to care what the underlying implementation is as long as you have the methods you need. This is useful when defining a class that requires some kind of authenticator:

```
class LoginValidator
{
      protected $adapter;

      public function setAuthenticator(
            AuthenticationAdapterInterface $adapter
      )
            {
                  $this->adapter = $adapter;
            }
}
```

## Strategy

The Adapter pattern might have many uses on its own, but it is often more useful when combined with the Strategy pattern. The difference is that the Adapter pattern defines the common interface, whereas the Strategy pattern implements one. The distinction between definition and implementation is key and makes for a natural pairing:

```
class AuthenticationAdapter
{
      protected $connection;

      public function authenticate($username, $password)
      {
            return $this->connection->authenticate($username, $password);
      }
                                                                    Continued
```

```
      public function setConnection(
            AuthenticationAdapterInterface $connection
      )
            {
                  $this->connection = $connection;
            }
}
```

Even though you are naming this an adapter, it is an implementation of the Strategy. With the previous Adapter pattern, you called each adapter individually. With the Strategy pattern, you let the AuthenticationAdapter do that for you:

```
$auth = new AuthenticationAdapter();
$auth->setConnection(new ISeriesAuthenticator ());
if ($auth->authenticate($username, $password)) {
      // logged in
}
```

It is a subtle, somewhat confusing, but important difference.

## Lazy Initialization and Lazy Loading

For many programming languages, Java® for example, it makes sense to preinitialize objects and relationships. Java is generally run as a long-lived process, and developers can use much of that up-front initialization to decrease the cost of initialization during runtime.

However, because PHP is a shared-nothing architecture and does not retain state in between requests, this kind of preinitialization can waste CPU cycles. As such, Lazy Initialization is a design pattern that is well worth your time to use as much as possible.

The way I see it, you have two ways of using Lazy Initialization in PHP. Although most languages will compile down to a binary that has all the required components either included or referenced, PHP does not do this. It starts with a clean slate for each request. Because of this, you must include dependencies again for each request. This typically looks as follows:

```
require_once 'config.php';
require_once 'classes/myclass.php';
require_once 'classes/anotherclass.php';
```

The effect is that myclass.php and anotherclass.php will be loaded regardless of whether they are needed. The work-around is to use the autoloader discussed in Chapter 1. In my testing, using the autoloader over static require() calls increased performance by more than 20 percent, even with an opcode cache.

The general approach for Lazy Initialization is to delay the creation of an object until it is needed, as the following example shows:

```
class Car
{
      protected $engine;

      public function getEngine()
      {
            if (!$this->engine instanceof Engine) {
                  $this->engine = new Engine();
            }
            return $this->engine;
      }
}
```

To retrieve the Engine object from the Car object, the getEngine() method must be called either from inside the class or externally. By doing this, the application will instantiate the Engine object only when getEngine() is called, even if the Car class has an instance. If the Engine object is heavy in terms of its initialization cost, the use of Lazy Initialization can prove beneficial.

Next is Lazy Loading. This design pattern is like Lazy Initialization, but you implement it from within the context of data instead of class instantiation. Although you can save CPU time by delaying the initialization of an object, you can save more time by delaying the loading of data.