Chapter 29

Popular among fans of Java and C++, design patterns are not a topic often discussed among PHP programmers. Yet, they are an important part of computer science. Furthermore, they apply to all programming languages.

Design patterns have their root in the work of Christopher Alexander in the context of designing buildings and cities. However, his work applies to any design activity, and it soon inspired computer scientists. The first popular book about software design patterns was *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. People commonly refer to them as the Gang of Four, or GoF.

29.1 Patterns Defined

Intuitively, we recognize patterns in our programming with almost every line of code. Given an array, you have a favorite idiom for looping over it. Since the `foreach` statement appeared in PHP, it's been my favorite.

From a larger perspective, we encounter the familiar problem of where to place functionality in PHP scripts. Most projects require dividing functionality into several modules. A flat, informational site benefits well from a simple scheme using headers and footers applied with `include` or `require`. Both examples have problems to be solved and memorized solutions. The conditions define a problem that has a known solution. Furthermore, after solving the problem a number of times, you gain an appreciation for the side effects, good and bad, of the solution.

The formal definition of design patterns includes four parts: a name, a description of the problem, a solution, and a set of consequences. The name gives us a convenient way to refer to the pattern. The problem description defines a particular set of conditions to which the pattern applies. The solution describes a best general strategy for resolving the problem. Finally, the pattern explains any consequences of applying the pattern.

Pattern solutions are not particularly fancy. They don't require the use of obscure features. They represent careful refinement over time, based on experience. They tend to optimize for reusability rather than efficiency. Naturally, a solution optimized for speed takes advantage of a particular situation and therefore is not well suited to the general case. For example, if you need the sum of three numbers, you can easily write them in a line of code. You would not use a general solution for the sum of 10,000 numbers, such as looping over an array.

Although patterns have their roots in building architecture, in the context of computer science they are closely linked to object-oriented design. Object-oriented programming aims to produce generalized software modules called objects. Design patterns seek to produce generalized solutions to common problems. This avoids the reinvention of the proverbial wheel.

Prior to PHP 5, PHP programmers found it difficult to implement design patterns efficiently in PHP. Thanks to PHP 5's revamped object model, design patterns are now easy to implement and are becoming a key ingredient in development of object-oriented PHP applications.

There are several advantages to using design patterns in your code. You don't need to think through the solution as long as you recognize that the problem matches the one solved by the pattern. You don't need to analyze the consequences of applying the pattern. You don't need to spend time optimizing the implementation.

Instead of having to come up with a solution, you only have to recognize what kind of problem you are facing. If the problem has an applicable design pattern, then you may be able to skip much of the design overhead and go directly to the implementation phase.

The consequences of using a certain design pattern are written in the pattern description. Instead of having to analyze the possible implications of using a certain algorithm—or worse, figure out why the algorithm you chose is not right for you after you implement it—you can refer to the pattern description. Implementing a solution from a design pattern gives you a fairly good idea about the complexity, limitations, and overhead of the solution.

The solutions supplied in design patterns tend to be efficient, especially in terms of reducing development and maintenance times. Simply put, you put other people's brains to work on your problem for free, which is a bargain.

If you've written large applications, it's quite possible that you would recognize similarities between some of the algorithms you used and the algorithms described in

certain design patterns. That is no coincidence—design patterns are there to solve real-world problems that you are likely to encounter regularly. It's quite possible that after performing a thorough investigation of a certain problem, the solution you came up with is similar to that in the design pattern. If you were aware of design patterns back then, it would have saved you at least some of the design time.

While this chapter is not meant to provide thorough coverage of design patterns, it acquaints you with some of the most popular ones and includes PHP implementation examples. If you're interested in further enhancing your knowledge of design patterns, definitely find a copy of the GoF book mentioned earlier. Craig Larman's *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process* is another well-recommended resource.

29.2 Singleton

Singleton is a design pattern that is useful when you want to create an object that should be accessible for different, distinct parts of your application. Especially if this object is supposed to contain large chunks of information, instantiating it over and over again may prove to be extremely inefficient. Instead, if you had a way of sharing the same instance between all of the different parts of the application, it would be ideal. Of course, global variables come to mind, but they require you to manage initialization. That is, you must make sure that nobody erases this variable by mistake, that nobody instantiates another instance of this class, and so forth. Relying on the application code to properly use the infrastructure is definitely not object-oriented. In object-oriented design, you would instantiate your own class to expose an API allowing you to take care of these things in the class itself instead of having to rely on every piece of application code to maintain system integrity.

Figure 29.1 shows the structure of a Singleton implementation in PHP.

Analyzing this class, you can spot three key features: a private, static property holding the single instance; a public, static method that returns the single instance; and a private constructor.

A private, static property holds a single instantiation of the class. As previously mentioned in the description of static class properties, static variables are similar to global variables. In this case, however, we take advantage of our ability to make this property private, thereby preventing application code from reading it or changing it.

A public, static method returns the only instantiation of the class. This single access point allows us to initialize the variable exactly once, before the application code accesses it. Thanks to its being static, we don't need to instantiate an object before we can call this method.

```
class Singleton
{
    static private $instance = NULL;

    private function __construct()
    {
        … perform initialization as necessary …
    }

    static public function getInstance()
    {
        if (self::$instance == NULL)
        {
            self::$instance = new Singleton();
        }

        return self::$instance;
    }

    … class logic goes here …
}
```

Figure 29.1   Singleton pattern.

The constructor is private. A Singleton class is one of the few situations in which it makes sense to use a private constructor. The private constructor prevents users from instantiating the class directly. They must use the `getInstance` method. Trying to instantiate the class using `$obj = new Singleton` will result in a fatal error, since the global scope may not call the private constructor.

One real-world example with which you can use the Singleton class is a configuration class, which wraps around your application's configuration settings. Listing 29.1 is a simple example. Thanks to the Singleton pattern, there's never more than one copy of the configuration file in memory. Any changes made to the configuration automatically persist.

Listing 29.1   *Configuration Singleton*

```
<?php
    /*
    ** Configuration file singleton
    */
    class Configuration
```

Listing 29.1   *Configuration Singleton (cont.)*

```
{
    static private $instance = NULL;
    private $ini_settings;
    private $updated = FALSE;
    const INI_FILENAME = "/tmp/corephp.ini";

    private function __construct()
    {
        if(file_exists(self::INI_FILENAME))
        {
            $this->ini_settings =
                parse_ini_file(self::INI_FILENAME);
        }
    }

    private function __destruct()
    {
        //if configuration hasn't changed, no need
        //to update it on disk
        if(!$this->updated)
        {
            return;
        }

        //overwrite INI file with the
        //version in memory
        $fp = fopen(self::INI_FILENAME, "w");
        if(!$fp)
        {
            return;
        }

        foreach ($this->ini_settings as $key => $value)
        {
            fputs($fp, "$key = \"$value\"\n");
        }

        fclose($fp);
    }

    public function getInstance()
    {
        if(self::$instance == NULL)
        {
            self::$instance = new Configuration();
        }
```

| Listing 29.1 | *Configuration Singleton (cont.)* |

```
        return self::$instance;
    }

    public function get($name)
    {
        if(isset($this->ini_settings[$name]))
        {
            return $this->ini_settings[$name];
        }
        else
        {
            return(NULL);
        }
    }

    public function set($name, $value)
    {
        //update only if different from what
        //we already have
        if(!isset($this->ini_settings[$name]) OR
            ($this->ini_settings[$name] != $value))
        {
            $this->ini_settings[$name] = $value;
            $this->updated = TRUE;
        }
    }
}

//Test the class
$config = Configuration::getInstance();
$config->set("username", "leon");
$config->set("password", "secret");
print($config->get("username"));
?>
```

29.3 Factory

Factory is a design pattern aimed at decoupling the instantiation of your objects from the application code that uses them. For example, you may want to use different kinds of objects depending on the situation. If you have two rendering classes, `HtmlRenderer` and `WmlRenderer`, and want your application to transparently use the right one depending on what kind of client is connected, you can easily do that using the Factory design pattern.

There are many different variants of the Factory design pattern. In Figure 29.2 we pick a simple one, which simply uses a global function.

```php
<?php
    //define abstract factory class
    class Renderer
    {
        private $document;

        abstract function render()
        {
        }

        function setDocument($document)
        {
            $this->document = $document;
        }
    }

    class HtmlRenderer extends Renderer
    {
        function render()
        {
            ... HTML rendering ...
        }
    }

    class WmlRenderer extends Renderer
    {
        function render()
        {
            ... WML rendering ...
        }
    }

    //Create the right kind of Renderer
    function RendererFactory()
    {
        $accept = strtolower($_SERVER["HTTP_ACCEPT"]);
        if(strpos($accept, "vnd.wap.wml") > 0)
        {
            return new WmlRenderer();
        }
        else
        {
            return new HtmlRenderer();
        }
    }

    //Application code
    $renderer = RendererFactory();
    $renderer->setDocument(…content…);
    $renderer->render();
?>
```

Figure 29.2 Factory pattern.

The Factory method receives no arguments, but in many situations you may wish to pass information to the Factory that will help it determine what kind of object should be instantiated. Nothing in the Factory pattern prevents you from passing arguments to the constructor.

A popular case for using factory methods is implementing an unserializer—a piece of code that takes a two-dimensional, serialized stream and turns it into objects. How do we write general-purpose code that will be able to instantiate any type of object that may appear in the stream? What if you want to specify different arguments to the constructor, depending on the type of object you're instantiating? Listing 29.2 contains an implementation.

| Listing 29.2 | *Registered classes with the Factory pattern* |

```php
<?php
   class Factory
   {
       private $registeredClasses = array();
       static private $instance = NULL;

       private function __construct() {}

       static function getInstance()
       {
           if(self::$instance == NULL)
           {
               self::$instance = new Factory();
           }
           return self::$instance;
       }

       function registerClass($id, $creator_func)
       {
           $this->registeredClasses[$id] = $creator_func;
       }

       function createObject($id, $args)
       {
           if(!isset($this->registeredClasses[$id]))
           {
               return(NULL);
           }
           return($this->registeredClasses[$id]($args));
       }
   }
```

| Listing 29.2 | *Registered classes with the Factory pattern (cont.)* |

```php
class MyClass
{
    private $created;
    public function __construct()
    {
        $created = time();
    }

    public function getCreated()
    {
        return($this->created);
    }
}

function MyClassCreator()
{
    return(new MyClass());
}

$factory = Factory::getInstance();
$factory->registerClass(1234, "MyClassCreator");
$instance = $factory->createObject(1234, array());
?>
```

Those of you who are familiar with the bits and bytes of PHP's syntax know that there's a simpler way of doing it. Listing 29.2 demonstrates a more object-oriented way to solve the problem, as it is done in other languages. It also allows for flexibility should you wish to implement additional logic in the creator (possibly sending some information to the constructor). In practice, it's accurate to say that PHP has built-in support for factory methods, utilized by simply writing `$object = new $classname`.

29.4 Observer

Observer is one of the most useful design patterns for developing large-scale object-oriented applications. It allows you, with the use of messages, to interconnect objects without their having to know anything about each other. At the heart of the Observer pattern are two main actors: observers and subjects. Observer objects find subject objects interesting and need to know when the subject changes. Typically, multiple observers monitor a single subject.

Listing 29.3 contains a simple implementation of the Observer pattern.

Listing 29.3	*Observer pattern*

```php
<?php
   interface Message
   {
      static function getType();
   };

   interface Observer
   {
      function notifyMsg(Message $msg);
   };

   class Subject
   {
      private $observers = array();

      function registerObserver(Observer $observer, $msgType)
      {
         $this->observers[$msgType][] = $observer;
      }

      private function notifyMsg(Message $msg)
      {
         @$observers = $this->observers[$msg->getType()];
         if(!$observers)
         {
            return;
         }

         foreach($observers as $observer)
         {
            $observer->notifyMsg($msg);
         }
      }

      function someMethod()
      {
         //fake some task
         sleep(1);

         //notify observers
         $this->notifyMsg(new HelloMessage("Zeev"));
      }
   }
```

| Listing 29.3 | *Observer pattern (cont.)* |

```
class HelloMessage implements Message
{
    private $name;

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

    function getMsg()
    {
        return "Hello, $this->name!";
    }

    static function getType()
    {
        return "HELLO_TYPE";
    }
}


class MyObserver implements Observer
{
    function notifyMsg(Message $msg)
    {
        if ($msg instanceof HelloMessage)
        {
            print $msg->getMsg();
        }
    }
}

$subject = new Subject();
$observer = new MyObserver();
$subject->registerObserver($observer,
    HelloMessage::getType());
$subject->someMethod();
?>
```

The beauty in the Observer pattern is that it allows subject objects to activate Observer objects without the subjects having any knowledge about the objects that observe them other than that they support the notification interface. The Observer pattern enables developers to connect dependent objects in different parts of the application, dynamically and as necessary, without having to provide specialized APIs

for each type of dependency. It also allows different Observer objects to select what kind of information interests them without having to change any code in the subject object.

One thing to worry about when implementing Observer is cyclic notification paths. An object may both observe other objects and be observed by other objects— that is, be both a Subject and an Observer. If two objects observe each other and deliver messages that trigger another message in their observing object, an endless loop occurs. In order to avoid it, it's best if you avoid delivering notification messages in your notification handler. If it's not possible, try to create a simple, one-sided flow of information, which will prevent cyclic dependencies.

29.5 Strategy

The Strategy pattern applies when you have a general problem to be solved by two or more algorithms. The choice of solutions represents a decision the user makes. For example, a graphics program allows for saving an image in many different formats, each with unique code for writing a file. The input to each of these routines is identical.

This pattern can also solve the problem of presenting a Web application in various languages or styles. Very simple schemes can get by with an array of translated words or colors for a theme, but complex customization may require code to produce dynamic results. I encountered this situation when trying to allow for international versions of an e-commerce site.

Aside from differences in language, people of the world format numbers differently. The `number_format` function goes a long way to solve this problem, of course. It doesn't address figures of money. Americans use $ to the left of numbers to represent dollars. Europeans may expect EUR, the symbol for a Euro. It's possible prices for Japanese customers should have yen to the right of the figure, depending on the situation.

To implement the strategy pattern, you must define a shared interface for all algorithms. You may then proceed with various implementations of this interface. In PHP we can implement this by defining a general class and extending it with subclasses. We can take advantage of polymorphism to promote a consistent interface to the functionality.

Listing 29.4 contains the base class, `localization`. It defines two methods, `formatMoney` and `translate`. The first method returns a formatted version of a money figure. The second method attempts to translate an English phrase into a local representation. The base class defines default functionality. Subclasses can choose to use the defaults or override them.

Listing 29.4	*Strategy pattern*

```php
<?php
    //Strategy superclass
    class Localization
    {
        function formatMoney($sum)
        {
            number_format($sum);
        }

        function translate($phrase)
        {
            return($phrase);
        }
    }
?>
```

Listing 29.5 contains an English subclass of localization. This class takes special care to place negative signs to the left of dollar signs. It doesn't override the translate method, since input phrases are assumed to be in English.

Listing 29.5	*English subclass*

```php
<?php
    //get Localization
    include_once('29-4.php');

    class English extends Localization
    {
        function formatMoney($sum)
        {
            $text = "";

            //negative signs precede dollar signs
            if($sum < 0)
            {
                $text .= "-";
                $sum = aba($sum);
            }

            $text .= "$" . number_format($sum, 2, '.', ',');

            return($text);
        }
    }
?>
```

Listing 29.6 contains a German subclass of `localization`. This class uses periods to separate thousands and commas to separate decimals. It also includes a crude `translate` method that handles only yes and no. In a realistic context, the method would use some sort of database or external interface to acquire translations.

Listing 29.6	*German subclass*

```php
<?php
    include_once('29-4.php');

    class German extends Localization
    {
        public function formatMoney($sum)
        {
            $text = "EUR " . number_format($sum, 2, ',', '.');

            return($text);
        }

        public function translate($phrase)
        {
            if($phrase == 'yes')
            {
                return('ja');
            }

            if($phrase == 'no')
            {
                return('nein');
            }

            return($phrase);
        }
    }
?>
```

Finally, Listing 29.7 is an example of using the `localization` subclasses. A script can choose between available subclasses based on a user's stated preference or some other clue, such as HTTP headers or domain name. This implementation depends on classes kept in files of the same name. After initialization, all use of the localization object remains the same for any language.

Listing 29.7	*Using `localization`*

```php
<?php
   print("Trying English<br>\n");
   include_once('29-5.php');
   $local = new English;
   print($local->formatMoney(12345678) . "<br>\n");
   print($local->translate('yes') . "<br>\n");

   print("Trying German<br>\n");
   include_once('29-6.php');
   $local = new German;
   print($local->formatMoney(12345678) . "<br>\n");
   print($local->translate('yes') . "<br>\n");
?>
```

One advantage of this pattern is the elimination of big conditionals. Imagine a single script containing all the functionality for formatting numbers in every language. It would require a switch statement or an if-else tree. It also requires parsing more code than you would possibly need for any particular page load.

Also consider how this pattern sets up a nice interface that allows later extension. You can start with just one localization module, but native speakers of other languages can contribute new modules easily. This applies to more than just localization. It can apply to any context that allows for multiple algorithms for a given problem.

Keep in mind that Strategy is meant for alternate functionality, not just alternate data. That is, if the only difference between strategies can be expressed as values, the pattern may not apply to the particular problem. In practice, the example given earlier would contain much more functionality differences between languages, differences which might overwhelm this chapter.

You will find the Strategy pattern applied in PEAR_Error, the error-handling class included in PEAR. Sterling Hughes wrote PEAR's error framework so that it uses a reasonable set of default behaviors, while allowing for overloading for alternate functionality depending on context.