Five common PHP design patterns

Jack D. Herrington   July 18, 2006

> Design patterns are just for Java™ architects -- at least that's what you may have been led to believe. In fact, design patterns are useful for everyone. If these tools aren't exclusive to architecture astronauts, what are they, and why are they useful in PHP applications? This article explains.

Editor's note: See the follow-up article "Five more PHP design patterns" to learn about five additional design patterns that you might find useful.

Design patterns were introduced to the software community in *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (colloquially known as the "gang of four"). The core concept behind design patterns, presented in the introduction, was simple. Over their years of developing software, Gamma et al found certain patterns of solid design emerging, just as architects designing houses and buildings can develop templates for where a bathroom should be located or how a kitchen should be configured. Having those templates, or *design patterns*, means they can design better buildings more quickly. The same applies to software.

Design patterns not only present useful ways for developing robust software faster but also provide a way of encapsulating large ideas in friendly terms. For example, you can say you're writing a messaging system to provide for loose coupling, or you can say you're writing an *observer*, which is the name of that pattern.

It's difficult to demonstrate the value of patterns using small examples. They often look like overkill because they really come into play in large code bases. This article can't show huge applications, so you need to think about ways to apply the principles of the example -- and not necessarily this exact code -- in your larger applications. That's not to say that you shouldn't use patterns in small applications. Most good applications start small and become big, so there is no reason not to start with solid coding practices like these.

Now that you have a sense of what design patterns are and why they're useful, it's time to jump into five common patterns for PHP V5.

The factory pattern

Many of the design patterns in the original *Design Patterns* book encourage *loose coupling*. To understand this concept, it's easiest to talk about a struggle that many developers go through in

large systems. The problem occurs when you change one piece of code and watch as a cascade of breakage happens in other parts of the system -- parts you thought were completely unrelated.

The problem is *tight coupling*. Functions and classes in one part of the system rely too heavily on behaviors and structures in other functions and classes in other parts of the system. You need a set of patterns that lets these classes talk with each other, but you don't want to tie them together so heavily that they become interlocked.

In large systems, lots of code relies on a few key classes. Difficulties can arise when you need to change those classes. For example, suppose you have a `User` class that reads from a file. You want to change it to a different class that reads from the database, but all the code references the original class that reads from a file. This is where the factory pattern comes in handy.

The *factory pattern* is a class that has some methods that create objects for you. Instead of using `new` directly, you use the factory class to create objects. That way, if you want to change the types of objects created, you can change just the factory. All the code that uses the factory changes automatically.

Listing 1 shows an example of a factory class. The server side of the equation comes in two pieces: the database, and a set of PHP pages that let you add feeds, request the list of feeds, and get the article associated with a particular feed.

Listing 1. Factory1.php

```php
<?php
interface IUser
{
  function getName();
}

class User implements IUser
{
  public function __construct( $id ) { }

  public function getName()
  {
    return "Jack";
  }
}

class UserFactory
{
  public static function Create( $id )
  {
    return new User( $id );
  }
}

$uo = UserFactory::Create( 1 );
echo( $uo->getName()."\n" );
?>
```
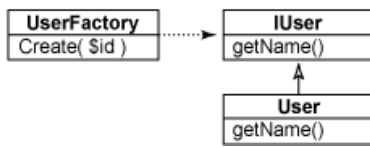
An interface called `IUser` defines what a user object should do. The implementation of `IUser` is called `User`, and a factory class called `UserFactory` creates `IUser` objects. This relationship is shown as UML in Figure 1.

Figure 1. The factory class and its related IUser interface and user class

If you run this code on the command line using the `php` interpreter, you get this result:

```
% php factory1.php
Jack
%
```

The test code asks the factory for a `User` object and prints the result of the `getName` method.

A variation of the factory pattern uses factory methods. These public static methods in the class construct objects of that type. This approach is useful when creating an object of this type is nontrivial. For example, suppose you need to first create the object and then set many attributes. This version of the factory pattern encapsulates that process in a single location so that the complex initialization code isn't copied and pasted all over the code base.

Listing 2 shows an example of using factory methods.

Listing 2. Factory2.php

```php
<?php
interface IUser
{
  function getName();
}

class User implements IUser
{
  public static function Load( $id )
  {
      return new User( $id );
  }

  public static function Create( )
  {
      return new User( null );
  }

  public function __construct( $id ) { }

  public function getName()
  {
    return "Jack";
  }
}

$uo = User::Load( 1 );
echo( $uo->getName()."\n" );
?>
```
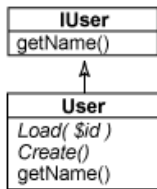
This code is much simpler. It has only one interface, `IUser`, and one class called `User` that implements the interface. The `User` class has two static methods that create the object. This relationship is shown in UML in Figure 2.

Figure 2. The IUser interface and the user class with factory methods

Running the script on the command line yields the same result as the code in Listing 1, as shown here:

```
% php factory2.php
Jack
%
```

As stated, sometimes such patterns can seem like overkill in small situations. Nevertheless, it's still good to learn solid coding forms like these for use in any size of project.

The singleton pattern

Some application resources are *exclusive* in that there is one and only one of this type of resource. For example, the connection to a database through the database handle is exclusive. You want to share the database handle in an application because it's an overhead to keep opening and closing connections, particularly during a single page fetch.

The singleton pattern covers this need. An object is a *singleton* if the application can include one and only one of that object at a time. The code in Listing 3 shows a database connection singleton in PHP V5.

Listing 3. Singleton.php

```php
<?php
require_once("DB.php");

class DatabaseConnection
{
  public static function get()
  {
    static $db = null;
    if ( $db == null )
      $db = new DatabaseConnection();
    return $db;
  }

  private $_handle = null;

  private function __construct()
  {
    $dsn = 'mysql://root:password@localhost/photos';
    $this->_handle =& DB::Connect( $dsn, array() );
  }

  public function handle()
  {
    return $this->_handle;
  }
}
```

```
print( "Handle = ".DatabaseConnection::get()->handle()."\n" );
print( "Handle = ".DatabaseConnection::get()->handle()."\n" );
?>
```

This code shows a single class called `DatabaseConnection`. You can't create your own `DatabaseConnection` because the constructor is private. But you can get the one and only one `DatabaseConnection` object using the static `get` method. The UML for this code is shown in Figure 3.

Figure 3. The database connection singleton

The proof in the pudding is that the database handle returned by the `handle` method is the same between two calls. You can see this by running the code on the command line.

```
% php singleton.php
Handle = Object id #3
Handle = Object id #3
%
```

The two handles returned are the same object. If you use the database connection singleton across the application, you reuse the same handle everywhere.

You could use a global variable to store the database handle, but that approach only works for small applications. In larger applications, avoid globals, and go with objects and methods to get access to resources.

The observer pattern

The observer pattern gives you another way to avoid tight coupling between components. This pattern is simple: One object makes itself observable by adding a method that allows another object, the *observer*, to register itself. When the observable object changes, it sends a message to the registered observers. What those observers do with that information isn't relevant or important to the observable object. The result is a way for objects to talk with each other without necessarily understanding why.

A simple example is a list of users in a system. The code in Listing 4 shows a user list that sends out a message when users are added. This list is watched by a logging observer that puts out a message when a user is added.

Listing 4. Observer.php

```
<?php
interface IObserver
{
  function onChanged( $sender, $args );
}

interface IObservable
{
  function addObserver( $observer );
```

```
}
class UserList implements IObservable
{
  private $_observers = array();

  public function addCustomer( $name )
  {
    foreach( $this->_observers as $obs )
      $obs->onChanged( $this, $name );
  }

  public function addObserver( $observer )
  {
    $this->_observers []= $observer;
  }
}

class UserListLogger implements IObserver
{
  public function onChanged( $sender, $args )
  {
    echo( "'$args' added to user list\n" );
  }
}

$ul = new UserList();
$ul->addObserver( new UserListLogger() );
$ul->addCustomer( "Jack" );
?>
```

This code defines four elements: two interfaces and two classes. The `IObservable` interface defines an object that can be observed, and the `UserList` implements that interface to register itself as observable. The `IObserver` list defines what it takes to be an observer, and the `UserListLogger` implements that `IObserver` interface. This is shown in the UML in Figure 4.

Figure 4. The observable user list and the user list event logger
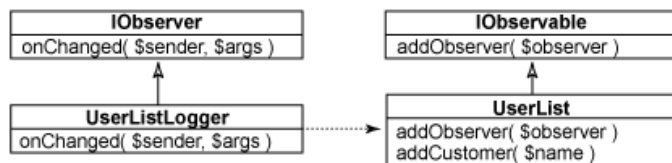
If you run this on the command line, you see this output:

```
% php observer.php
'Jack' added to user list
%
```

The test code creates a `UserList` and adds the `UserListLogger` observer to it. Then the code adds a customer, and the `UserListLogger` is notified of that change.

It's critical to realize that the `UserList` doesn't know what the logger is going to do. There could be one or more listeners that do other things. For example, you may have an observer that sends a message to the new user, welcoming him to the system. The value of this approach is that the `UserList` is ignorant of all the objects depending on it; it focuses on its job of maintaining the user list and sending out messages when the list changes.

This pattern isn't limited to objects in memory. It's the underpinning of the database-driven message queuing systems used in larger applications.

The chain-of-command pattern

Building on the loose-coupling theme, the *chain-of-command* pattern routes a message, command, request, or whatever you like through a set of handlers. Each handler decides for itself whether it can handle the request. If it can, the request is handled, and the process stops. You can add or remove handlers from the system without influencing other handlers. Listing 5 shows an example of this pattern.

Listing 5. Chain.php

```php
<?php
interface ICommand
{
  function onCommand( $name, $args );
}

class CommandChain
{
  private $_commands = array();

  public function addCommand( $cmd )
  {
    $this->_commands []= $cmd;
  }

  public function runCommand( $name, $args )
  {
    foreach( $this->_commands as $cmd )
    {
      if ( $cmd->onCommand( $name, $args ) )
        return;
    }
  }
}

class UserCommand implements ICommand
{
  public function onCommand( $name, $args )
  {
    if ( $name != 'addUser' ) return false;
    echo( "UserCommand handling 'addUser'\n" );
    return true;
  }
}

class MailCommand implements ICommand
{
  public function onCommand( $name, $args )
  {
    if ( $name != 'mail' ) return false;
    echo( "MailCommand handling 'mail'\n" );
    return true;
  }
}

$cc = new CommandChain();
$cc->addCommand( new UserCommand() );
$cc->addCommand( new MailCommand() );
$cc->runCommand( 'addUser', null );
$cc->runCommand( 'mail', null );
```
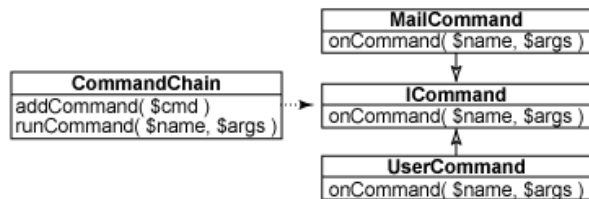
```
?>
```

This code defines a `CommandChain` class that maintains a list of `ICommand` objects. Two classes implement the `ICommand` interface -- one that responds to requests for mail and another that responds to adding users. The UML is shows in Figure 5.

Figure 5. The command chain and its related commands

If you run the script, which contains some test code, you see the following output:

```
% php chain.php
UserCommand handling 'addUser'
MailCommand handling 'mail'
%
```

The code first creates a `CommandChain` object and adds instances of the two command objects to it. It then runs two commands to see who responds to those commands. If the name of the command matches either `UserCommand` or `MailCommand`, the code falls through and nothing happens.

The chain-of-command pattern can be valuable in creating an extensible architecture for processing requests, which can be applied to many problems.

The strategy pattern

The last design pattern we will cover is the *strategy* pattern. In this pattern, algorithms are extracted from complex classes so they can be replaced easily. For example, the strategy pattern is an option if you want to change the way pages are ranked in a search engine. Think about a search engine in several parts -- one that iterates through the pages, one that ranks each page, and another that orders the results based on the rank. In a complex example, all those parts would be in the same class. Using the strategy pattern, you take the ranking portion and put it into another class so you can change how pages are ranked without interfering with the rest of the search engine code.

As a simpler example, Listing 6 shows a user list class that provides a method for finding a set of users based on a plug-and-play set of strategies.

Listing 6. Strategy.php

```
<?php
interface IStrategy
{
  function filter( $record );
}

class FindAfterStrategy implements IStrategy
{
  private $_name;
```

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

  public function filter( $record )
  {
    return strcmp( $this->_name, $record ) <= 0;
  }
}

class RandomStrategy implements IStrategy
{
  public function filter( $record )
  {
    return rand( 0, 1 ) >= 0.5;
  }
}

class UserList
{
  private $_list = array();

  public function __construct( $names )
  {
    if ( $names != null )
    {
      foreach( $names as $name )
      {
        $this->_list []= $name;
      }
    }
  }

  public function add( $name )
  {
    $this->_list []= $name;
  }

  public function find( $filter )
  {
    $recs = array();
    foreach( $this->_list as $user )
    {
      if ( $filter->filter( $user ) )
        $recs []= $user;
    }
    return $recs;
  }
}

$ul = new UserList( array( "Andy", "Jack", "Lori", "Megan" ) );
$f1 = $ul->find( new FindAfterStrategy( "J" ) );
print_r( $f1 );

$f2 = $ul->find( new RandomStrategy() );
print_r( $f2 );
?>
```
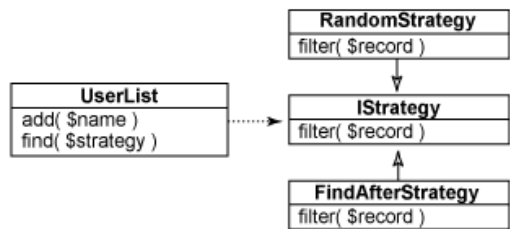
The UML for this code is shown in Figure 6.

Figure 6. The user list and the strategies for selecting users

The `UserList` class is a wrapper around an array of names. It implements a `find` method that takes one of several strategies for selecting a subset of those names. Those strategies are defined by the `IStrategy` interface, which has two implementations: One chooses users randomly and the other chooses all the names after a specified name. When you run the test code, you get the following output:

```
% php strategy.php
Array
(
    [0] => Jack
    [1] => Lori
    [2] => Megan
)
Array
(
    [0] => Andy
    [1] => Megan
)
%
```

The test code runs the same user lists against two strategies and shows the results. In the first case, the strategy looks for any name that sorts after `J`, so you get Jack, Lori, and Megan. The second strategy picks names randomly and yields different results every time. In this case, the results are Andy and Megan.

The strategy pattern is great for complex data-management systems or data-processing systems that need a lot of flexibility in how data is filtered, searched, or processed.

Conclusions

These are just a few of the most common design patterns used in PHP applications. Many more are demonstrated in the *Design Patterns* book. Don't be put off by the mystique of architecture. Patterns are great ideas you can use in any programming language and at any skill level.

Related topics

- Design patterns on Wikipedia
- Observer patterns on the C2 Wiki design
- Design Patterns: Elements of Reusable Object-Oriented Software
- Head First Design Patterns
- PHP Hacks: Tips & Tools For Creating Dynamic Websites

© Copyright IBM Corporation 2006
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)