PHP5 Design Patterns in a Nutshell

By Janne Ohtonen, August 2006

Contents

Introduction

This document has PHP5 design patterns described shortly. Reader can check out quickly which kind of design patterns are available and what is their nature and usage. Then it is possible to get more information about needed design patterns for example with Google (links are not provided here, because they can get out of order easily).

As a reading, author suggests at least these books:

phplarchitects's Guide to PHP Design Patterns by Jason E. Sweat
Design Patterns – Elements of Reusable Object-Oriented Software by Gamma et all

From Internet it is possible to find more information at least from these sites:

http://fi.php.net
http://www.phppatterns.com

Notice, that most of these design patterns apply also to other programming languages than PHP5 but in this document all the code example are written for PHP.

Acknowledgments

This document is adapted from the book phplarchitects's Guide to PHP Design Patterns by Jason E. Sweat and several documents found from Internet by Janne Ohtonen.

The Active Record Pattern

This is the simplest design pattern for handling database related things with PHP. It interacts with the database directly and simply. There are other design patterns too to use for database interaction: Table Data Gateway pattern and Data Mapper. You should check also PHP interface to database called PDO (http://fi.php.net/pdo).

This is possibly the simplest way to refactor your code from procedural programming to object-oriented. You just make a class that handles all the database interaction and then use it in your application. This can cause in bigger projects cumbersome database classes with lots of code.

Example code for the Active Record pattern:

```
class DatabaseHandlerForExampleClass {
        Private $connection;
        Private $field1;
        Private $field2;
        Public function __construct($conn) { $this->connection = $conn; }
        Public function save() { /* save row into table */ }
}
```

The Adapter Pattern

The Adapter pattern can be used to separate third-party components and objects from your code. That can be very useful especially in situations where the 3rd party components ought to be changed in the future. Naturally, you want to avoid changing the application code, because that can lead to incompatibilities with other code, so Adapter pattern can be used in-between the application and 3rd party components to avoid such situations.

Example code for the Adapter pattern:

```
class AdapterForThirdPartyComponentClass {
        Private function __construct() { }
        Public static function getThirdPartyComponent($version = false) {
                Switch($version) {
                        Case “version2”: return newThirdPartyComponentVer2();
                        Default: return newThirdPartyComponent();
                }
        }
}
```

Of course, you have many ways to actually realize Adapter pattern, so the code example above is just one suggestion but it gives the idea of desired behavior. This pattern is mainly meant for single components or a small amount of them. Wrapping bigger amount of components can be achieved with Façade pattern.

The Data Mapper Pattern

Since both Active Record and Table Data Gateway patterns are very tightly coupled with the data itself and it causes changes in database to affect code in application, the Data Mapper pattern is designed to detach the application code from the database structure. This way changes in database won't necessary affect the code immediately.

The key point in Data Mapper is the mapper itself that maps the database data to application data. This can be achieved by many means like hard coding the mapping into mapper or putting the mapping into some XML file and reading from it.

Example code for the Data Mapper pattern:

```
class ExampleDataMapper {
        Private $connection;
        Public function __construct ($conn) { $this->connection = $conn; }
        Public function save($dataObject) { /* saves the data object*/ }
        Public function delete($dataObject) {
                /* deletes from database according to data object*/
        }
}
```

The Decorator Pattern

If you end up into subclass mess where you have several subclasses, then you should consider using Decorator pattern. It is a good alternative for sub classing especially when subclasses have same base class. Decorator pattern also helps to avoid repetition of the code and great number of subclasses.

Decorator class sort of enhances the actual class which is missing some properties that need to be added. Also good think in Decorator classes is that you can chain them and get that way the end result. Decorators do not care about the subclasses because they have their own implementation. So, Decorators have in a way a lot of common with Proxies.

Example code for the Decorator Pattern:

```
interface SomeBaseInterface {
        public function doSomething();
}
class SomeBaseClass implements SomeBaseInterface {
```

```
        public function __construct() {}
        public function doSomething() { return $this->doSomethingPrivate() . '…'; }
        private function doSomethingPrivate() {
                return 'Test text from SomeBaseClass';
        }
}
class SomeBaseClassDecorator {
        protected $baseClass;
        public function __construct($base) {$this->baseClass = $base; }
        public function doSomething() { $this->baseClass->doSomething(); }
}
class ExtendedSomeBaseClassDecorator extends SomeBaseClassDecorator {
        protected $moreText = 'More!!!';
        public function __construct($more, $someBaseInterfaceExtender) {
                $this->moreText = $more;
                parent($someBaseInterfaceExtender);
        }
        public function doSomething() {
                return $this->baseClass->doSomething() . $this->moreText;
        }
}
```

As you can see from the example code, the original text ‘Test text from SomeBaseClass’ comes out in the end as ‘Test text from SomeBaseClass…More!!!’. If there was more changes needed then it is possible just to extend the ExtendedSomeBaseClassDecorator and put in the new behavior.

The Façade Pattern

The Façade pattern is for providing simple interface to more complex subsystems. It helps also to hide the implementation behind the Façade so that the client objects do not need to see all their complexity. Another good reason is to separate the subsystem from the client code, so that it is easier to replace and update the subsystem, especially if it is 3rd party (compare to the Adapter pattern). It does not prevent the clients that need more complex connections to subsystem from doing it because it just provides simpler interface for the subsystem as addition.

The Factory Pattern

The Factory pattern is designed for encapsulating the instantiation of objects. This way it is possible to put all the details that are involved with creating a new object into one place and hidden from the usage of the object. Some classes may be very complicated to instantiate and it would be very unwise to put that instantiation code in each place where the object is needed. Therefore it is better to encapsulate the instantiation and concentrate the code into one place.

Example code for the Factory pattern:

$connection = new mysql_connection('user', 'pwd', 'db'); →
$connection = $this->_getConnection();

For example in the class above, if the database connection is taken in several places, then the username and password would also in several places without Factory design pattern. When that information is centralized into one place, then for example change of password is very easy to realize in code, because it needs to be put only into one place.

The Iterator Pattern

Iterator pattern is intended for handling some amount of objects in collections without revealing the actual implementation of the collection. It is very useful to keep collections of objects in different situations and therefore there is a need for handling these collections. In PHP there is very profound collection of functions available for handling arrays that are collections of objects. So, that is a good reason for using them as base for iterators in PHP.

To make your own iterator you have to have a Factory for the collection (see the Factory pattern) which creates a new instance of the iterator. The iterator has to implement the interface for first(), next(), currentItem() and isDone() which are used to handle advancing in the collection.

Example code of the Iterator pattern:

```
class CustomIterator {
        Protected $objectCollection;
        Public function __construct () { $this->objectCollection = array(); }
        Public function first() { reset($this->objectCollection);  }
        Public function next() { return next($this->objectCollection); }
        Public function currentItem() { return current($this->objectCollection); }
        Public function isDone() { return false === currentItem(); }
}
```

It is also possible to make Variant Iterator, which has next, currentItem and isDone put together into one method next(). There are many ways to realize that kind of behavior with PHP. Filtering Iteration means that you have somehow selected which items to represent from the collection (for example you can give objects types and have many collections inside the iterator). Sorting Iterator gives capability to give out the elements from the iterator in a certain order.

PHP5 has also a Standard PHP Library Iterator, so called SPL Iterator. It is very powerful way for iterating through collections. The interface for the SPL Iterator is little bit different: you have to implement five functions to use it: current(), next(), key(), valid()

and rewind().

Example code for the SPL Iterator:

```
class CustomSPLIterator {
        Protected $objectCollection;
        Protected $valid;
        Public function __construct () { $this->objectCollection = array(); }
        Public function rewind() {
                $this->valid = (false !== reset($this->objectCollection));
        }
        Public function next() {
                $this->valid = (false !== next($this->objectCollection));
        }
        Public function current() { return current($this->objectCollection); }
        Public function key() { return key($this->objectCollection); }
        Public function valid () { return $this->valid; }
}
```

The Mock Object Pattern

This design pattern is intended to help testing and developing objects that depend on other objects that do not exist yet. It is very handy for developing one part of the application which is dependent on the other. The idea is to make Mock Object that has the same interface than the actual object but does not contain the actual realization of the object but has some simple pre-programmed responses and possible debug code.

Imagine a situation where you have divided the application in to two parts and you are realizing the other part and someone else is realizing the other. You have agreed the interfaces that you use and for some reason you develop the application a bit faster than the other. Then you make Mock Objects that simulate the other person's objects behind the interface and you are able to test and develop your application despite the other half. Then when you get the real objects, you just replace your Mock Objects with the real ones.

The Model-View-Controller Pattern

This design pattern is meant for separating you application business logic from the views that show the results and ask for the input from user. The application is divided into three parts: model, view and controller. Each part has a specific task to do. This pattern is more like conceptual guideline than actual coding pattern. So, let's go through each part from the MVC pattern:

Model is the brains of the application. It contains the code for business logic and more complex operations in application. Both View and Controller may ask Model for

information or processing, so it acts also as information medium. Model does not have any presentation- or request related features. There can be different kinds of Models: Simple Domain Model and Rich Domain Model

Controller takes in all the requests and controls the Views that show the result. It may ask from Model more information or processing and then according to result show different Views.

View is the actual view that is shown to user. It may ask information from the Model to show. Views can be normal views or Template Views when templates are used and processed through template engines.

The Mono State Pattern

This design pattern is meant for using in situations where objects need to have same common global state meaning that each instance from same object returns the same information in each instance. This feature can be used for example in settings classes.

Example code for the Mono State pattern:

```
Class ApplicationConfig  {
        Private $state;
        Public function __contruct() {
                $key = '_ApplicationConfig_state_key_';
                If(!isset($_SESSION[$key]) or !is_array($_SESSION[$key])) {
                        $_SESSION[$key] = array();
                }
                $this->state = $_SESSION[$key];
        }
        public function setState($newState) { $this->state = $newState; }
        public function getState() { return $this->state; }
}
```

As you can see from the example code, the object attribute ‘state’ is distributed to all instance through usage of PHP Session. Since the ‘state’ attribute is actually a reference to an attribute in Session, the same value is seen in each instance. So, Mono State can be seen as one special kind of Singleton pattern.

The Observer Pattern

This pattern is used to allow connections between objects that are interested to each other. The Observable class notifies the Observer classes when change occurs. The Observers can react to change as they are designed. One good side is also that Observers can connect and disconnect to Observable class as they wish, so this way Observable class is not responsible for the actions that Observers take.

Example code for the Observer pattern:

```
class TestObservable {
        Protected $observers = array();
        Public function attachObserver($observer) { $this->observers[] = $observer; }
        Public function detachObserver($observer) {/* remove observer from array*/ }
        Public function notifyObservers() { /* call ->update($this) for each observer */ }
}
class TestObserver {
        public function update($testobservable) {
                /* Do whatever observer does with testobservable */
        }
}
```

Of course you need also the code that attaches the observer to observable, etc.

The Proxy Pattern

Proxy is located between the subject and client objects. It can have some functionality like enhanced security (protection proxy), connections to remote site (remote proxy), lazy instantiation of subject (Lazy Instantiating Proxy) or connections to local resources (virtual proxy). In its simplest form it just passes through the calls from client object to subject object.

Example code for the Proxy pattern:

```
class Subject {
        Public function __construct() { }
        Public function doSomething() {}
}
class ProxySubject {
        protected $subject;
        Public function __construct() { $this->subject = new Subject(); }
        Public function doSomething() {
                /* for example first check the security, then do the subject */
                $this->subject->doSomething();
        }
}
```

Dynamic Proxy is PHP5 specific way to quickly get a proxy without writing each method. It is done using the __call() method which redirects all the calls for subject through proxy. Then it is possible to implement only those methods that actually need the proxy functionality.

Example code for the Dynamic Proxy:

```
class DynamicProxyShowCase {
        protected $subject;
        Public function __construct($subject) { $this->subject = $subject; }
        Public function __call($method, $args) {
                Return call_user_func_array(array($this->subject, $method), $args);
        }
}
```

The Registry Pattern

The Registry pattern is designed for accessing several objects through one object which works as a registry for the other objects. There are different ways to implement this behavior but arrays are very efficient in PHP and they can function as a base for object registry.

Example code for the Registry pattern:

```
class Registry {
        Private $store = array();
        Private function __construct() {}
        Public  function isValid($key) { return array_key_exists($key, $this->store); }
        Public function get($key) {
                If($this->isValid($key)) return $this->store[$key];
                return null;
        }
        public function set($key, $obj) { $this->store[$key] = $obj; }
        public function getInstance() {
                static $instance = array();
                if(!$instance) $instance[0] = new Registry();
                return $instance[0];
        }
}
```

The Singleton Pattern

This design pattern is used to get always the same instance from the class. This can be used to avoid global attributes and creating new instances from the class without obvious benefit. Usually Singleton objects have private constructor and the instance is got through a method designed for getting it.

Example code for the Singleton pattern:

```
Class ExampleSingletonClass {
        Private $instance;
```

```
        Private function __construct() { }
        Public function getInstance() {
                If(!$this->instance) $this->instance = new ExampleSingletonClass ();
                return $this->instance;
        }
}
```

You have to be careful when using Singleton classes, that you don't lose the original idea behind the design pattern.

The Specification Pattern

The Specification pattern is designed for encapsulating business logic, so that it is easier to adapt and reuse. The idea is to either validate the object to satisfy certain criteria or to identify specific objects from a collection with given criteria. There are many ways to do this specification and it can be divided into three sections called Hard Coded Specification, Parameterized Specification and Composite Specification.

The Hard Coded Specification means that the condition that satisfies the object criteria is hard coded into specification object. For example if you have to get people older than 40 years old who are female, then you would code that into specification class. In Parameterized Specification you can give the criteria for the specification class that is used to search the people that have the given age and gender. So, it makes the specification class a little bit more usable but on the other hand also a little bit more complicated.

Example code for the Specification pattern:

```
class ExampleHardCodedSpecification {
        Public function isSatisfiedBy($criteria) {
                Return $criteria->condition1 >= 23;
        }
}
class ExampleParameterizedSpecification {
        Protected $precondition;
        Public function __construct($precond) {
                $this->precondition = $precond;
        }
        public function isSatisfiedBy ($criteria) {
                Return $criteria->condition1 >= $this->precondition;
        }
}
```

The State Pattern

Difference between the State and Strategic patterns is that in State pattern you change the behavior of the object according to its state where in Strategic pattern the behavior is set in the construction phase. So, State pattern is more dynamic than the Strategic.

Example code for the State pattern:

```
class Human {
	Private $currentGoingMethod = 'Car';
	private $feelsLike = 'going';
	public function doWhatHumanFeelsLike() {
		If($this->feelsLike == 'going') {
			$goMethod = new $this->currentGoingMethod . 'Class';
			$goMethod->go();
		}
	}
	public function  setCurrentGoingMethod($method) {
		$this->currentGoingMethod = $method;
	}
}
```

So, if the current going method is set to Car, then the Human goes by car. And the state of the going method could be changed for example through setCurrentGoingMethod method.

The Strategy Pattern

This pattern is designed for simplifying the code in cases where the object has to act several ways depending on its use. In Strategy Pattern the object's use is specified in construction phase.

For example if you have a object called Human and it has method go. Then you have several choices for going by car, by walk, etc. Then you could have method go($method) and then inside the method you could have switch case that does what ever that method does for each type separately. But you can also make new class for each method and then in go -method call for the one that you are using.

Example code for the Strategy pattern:

```
class Human {
	Public function go($method = 'Car') {
		$goMethod = new $method . 'Class'; //new CarClass or WalkClass
		$goMethod->go(); //Do the actual going instead of messy switch-case
	}
}
```

So, instead of having a messy switch case where the actual 'going' is done, you now instantiate a new class which is meant for this purpose. This way, if there are some changes in that method, you find it easily and won't affect other code by accident. Of course the class has to have appropriate interface.

The Table Data Gateway Pattern

Difference to the Active Record pattern is that this pattern focuses more on tables, which are collections of rows (tuples) when the Active Record pattern focuses on rows as the name also implies. Thus, this abstracts a single table when the Active Record pattern abstracts a single row.

Example code for the Table Data Gateway pattern:

```
class SomeTableGateway {
        Private $connection;
        Public function __construct($conn) { $this->connection = $conn; }
        Public function getAll() { /* select * from table and return as array */  }
        Public function add($field1, $field2) { /* insert fields into table */ }
}
```

The Value Object Pattern

This design pattern is intended to make the object works as immutable. It means that the original object is never changed after initialization and when changes are done, new instance is returned. The values of the attributes in object are set through constructor and there does not exist any setters (getters may be, since they do not change anything).

Most important things for creating an immutable object are:

1. Protect the attributes so, that it is not possible to set them directly.
2. Set the attributes in constructor
3. There cannot be any setters or other functions that change the values of attributes without returning new instance

Example code for the Value Object pattern:

```
Public function add($newValue) { $this->amount += $newValue; } →
Public function add($more) { return new ExampleClass($this->amount + $more); }
```

The Visitor Pattern

This pattern can be seen as a dependency inverse of the Strategy pattern. It means that the visited component is not aware of the actual implementation behind some functionality that is described with interface. The component only knows it has been passed a visitor object and that it must call for particular method. Thus, the Visitor pattern takes the

components off the context and encapsulates the context-specific functionality within visitor implementations.

Example code for the Visitor pattern:

```
abstract class GoingMethodComponent {
        Public function visit (GoingMethodComponent $comp) {
                $method = 'visit' . get_class($comp);
                $this->$method($comp);
}
public abstract function visitCarClass ();
}
class CarClass extends GoingMethodComponent {
        public function __construct() { /* Do what car class constructor does */ }
        public function visitCarClass() { /* This is called to do what car class does */ }
}
```