**Design Patterns**

**Introduction**

A design pattern, simply put, is a recommended best practice for solving a particular problem. A design pattern is made up of 4 pieces:

* Its name
* A discussion of the problem
* A discussion of the solution
* The consequences (pros & cons)

**The singleton pattern**

The singleton pattern is a *creational* pattern that will restrict an application to creating only a single instance of a particular class type. E.g. a website will need a DB connectivity object, but should have only one! So you could use a singleton to enforce that restriction.

Use a static attribute to guarantee that only one instance of a particular class exists:

Class SomeClass {

Static private $\_instance = null;

}

Now, if you store the actual instance in that attribute (i.e. assign an object to it), then all references to this class can use that attribute.

The next step is to create a method that will create an instance of the class if one does not exist and return the instance regardless:

Class SomeClass {

Static private $\_instance = null;

Static function getInstance() {

If(self::$\_instance === null) {

Self::$\_instance = new SomeClass();

}

Return self::$\_instance;

}

}

It is common for a singleton to name this method *getInstance.*

Now the class can be used in this manner:

$obj1 = SomeClass::getInstance();

If a user creates a second object, the same instance as $obj1 will be returned ($obj2 = SomeClass::getInstance()).

*However,* if a user tries to create a *new* object of that class type using *new* or *clone*, you will end up with multiple instances, defeating the purpose of a singleton. The trick to preventing that from happening is to make a private constructor that does nothing. Now, the following code will trigger an error:

$obj = new ClassName();

Another good use for a singleton is to create one global object, such as a configuration object used by an entire site:

Class Config {

Static private $\_instance = null;

Private $\_settings = [];

//private methods cannot be called

Private function \_\_construct() {}

Private function \_\_clone() {}

Static function getInstance() {

If(self::$\_instance === null) {

Self::$\_instance = new Config();

}

}

//method for defining setting settings:

Function set($index, $value) {

$this->settings[$index] = $value;

}

//method for getting a setting

Function get($index) {

Return $this->settings[$index];

}

}

//create object

$CONFIG = Config::getInstance();

$CONFIG->set(‘live’, ‘true’);

Echo $CONFIG->get(‘live’);

Unset($CONFIG);

**The factory pattern**

The factory pattern is another *creation* pattern (like singleton), but unlike singleton, which creates and manages a single object of a single class type, the factory pattern is used to manufacture potentially multiple objects of many different class types.

The factory pattern becomes useful in situations where the type of object that needs to be generated isn’t known when the program is written but only once the program is running.

Another clue for when the factory pattern might be appropriate is when there’s an abstract base class, and different derived subclasses will need to be created on the fly.

The factory pattern works via a static method, conventionally named *Create(), Factory(), FactoryMethod(),* or *CreateInstance()*. The method takes at least one argument, which indicates the type of object to create. The method then returns an object of that type.

Abstract class ShapeFactory {

//static method that creates objects:

Static function Create($type, array $sizes) {

//determine the object type based on the parameter

Switch($type) {

Case ‘Rectangle’:

Return new Rectangle($sizes[0], $sizes[1]);

Break;

Case ‘Triangle’:

Return new Triangle($sizes[0], $sizes[1], $sizes[2]);

Break;

}

}

}

//for now, just *Rectangle* and *Triangle* classes are known, but there is room to add more when needed.

To use *ShapeFactory:*

$obj = ShapeFactory::Create(‘Rectangle’, $dimensions);

//now you can use the methods from these specific classes.

\*note the capital letter on the *Create()* method\*

**The composite pattern**

The singleton and factory pattern are *creational*: use to generate one or more objects. Another category of pattern is *structural*. These patterns apply in situations where a non-traditional class structure, or a modification of an existing class structure, is required.

The composite pattern applies in situations where you have an object that might represent a single entity or a composite entity, but still needs to be useable in the same manner.

To implement the composite pattern, you’ll normally start with an abstract base class that will be extended by the different subclasses. The base class needs to identify the methods for adding and removing “*leaves”* (composite items). The base class also needs to identify any functionality that the composite, or its sub-elements, need to do:

Abstract class FormComponent {

Abstract function add($obj); //key method to composite

Abstract function remove($obj); //key method to composite

Abstract function display(); //specific functionality needed by this e.g

Abstract function validate(); //specific functionality needed by this e.g

Abstract function showError(); //specific functionality needed by this e.g

Each class now inherits from the derived class. By definition, each must also define the implementation of the abstract methods:

Class Form extends FormComponent {

Private $\_elements = [];

Function add(FormComponent $obj) {

$this->\_elements[] = $obj;

}

Function display() {

//display the entire form

}

}

Class FormElement extends FormComponent {

Function add(FormComponent $obj) {

Return $obj; //or false

}

Function display() {

//display element

}

}

$form = new Form();

$email = new FormElement();

$form->add($email);

Example

Abstract class WorkUnit {

//for storing work to be done:

Protected $tasks = [];

//for storing the employee or team name:

Protected $name = null;

//constructor assigns the name:

Function \_\_construct($name) {

$this->name = $name;

}

//method that returns the name

Function getName() {

Return $this->name;

}

//abstract functions to be implemented:

Abstract function add(Employee $e);

Abstract function remove(Employee $e);

Abstract function assignTask($task);

Abstract function completeTask($task);

}

Class Team extends WorkUnit {

//for storing team members:

Private $\_employees = [];

//implement the abstract methods…

Function add(Employee $e) {

$this->\_employees[] = $e;

Echo “<p>{$e->getName()} has been added to team {$this->getName()}.</p>”;

}

Function remove(Employee $e) {

$index = array\_search($e, $this->\_employees);

Unset($this->\_employees[$index]);

Echo “<p>{$e->getName()} has been removed from team {$this->getName()}.</p>”;

}

Function assignTask($task) {

$this->tasks[] = $task;

Echo “<p>A new task has been assigned to team {$this->getName()}. It should be easy to do with {$this->getCount()} team member(s).</p>”;

}

Function completeTask($task) {

$index = array\_search($task, $this->tasks);

Unset($this->tasks[$index]);

Echo “<p>The ‘$task’ task has been completed by team {$this-getName()}.</p>”;

}

//method for returning the number of team members:

Function getCount() {

Return count($this->\_employees);

}

}

Class Employee extends WorkUnit {

//empty functions:

Function add(Employee $e) {

Return false;

}

Function remove(Employee $e) {

Return false;

}

//implement the abstract methods:

Function assignTasks($task) {

$this->tasks[] = $task;

Echo “<p>A new task has been assigned to {$this->getName()}. It will be done by {$this->getName()} alone.</p>”;

}

Function completeTask($task) {

$index = array\_search($task, $this->tasks);

Unset($this->tasks[$index]);

Echo “<p>The ‘$task’ task has been completed by employee {$this->getName()}.</p>”;

}

}

//HTML

<?php

Require(‘WorkUnit.php’);

//create the objects

$alpha = new Team(‘Alpha’);

$john = new Employee(‘John’);

$ali = new Employee(‘Ali’);

//assign employees to the team

$alpha->add($john);

//assign tasks:

$alpha->assignTask(‘Do something great.’);

$ali->assignTask(‘Do something grand.’);

//complete a task

$alpha->completeTask(‘Do something great’);

//remove a team member

$alpha->remove($john);

//delete the objects:

Unset($alpha, $john, $ali);

?>

Remember, the composite pattern allows one class type to be used like another, even if one is composed of the other class type.

**The strategy pattern**

The strategy pattern is a type of *behavioural* pattern. *Behavioural* patterns are used to address how an application runs. As a comparison, the factory pattern can change the object type on the fly, but the strategy pattern can change an algorithm on the fly (an algorithm just being a process or set of code used to perform a calculation or solve a problem). *Strategy* is most useful in situations where you have classes that may be similar, but not related, and differ only in their specific behaviour.

E.g. Say you need a filtering system for strings. Different filters might include:

* Stripping out HTML
* Crossing out swear words
* Catching character combinations that can be used to send spam through contact forms and the like

The only thing that these 3 approaches have in common is that they’re all applied to strings.

First start by defining an interface that dictates the needed functionality:

Interface Filter {

Function filter($str);

}

Specific filter types then implement their specific versions of the interfaces method:

Class HTMLFilter implements Filter {

Function filter($str) {

//strip out HTML here

Return $str;

}

}

Class SwearFilter implements Filter {

Function filter($str) {

//cross out swear words

Return $str;

}

}

Finally, another class would be written that could use any filter:

Class FormData {

Private $\_data = null;

Function \_\_construct($input) {

$this->\_data = $input;

}

Function process(Filter $type) {

$this->\_data = $type->filter($this->\_data);

}

}

$form = new FormData($someUserInput);

$form->process(new HTMLFilter());

Example

Interface iSort {

Function sort(array $list);

}

//the MultiAlphaSort sorts a multidimensional array alphabetically.

Class MultiAlphaSort implements iSort {

//how to sort:

Private $\_order;

//sort index:

Private $\_index;

//constructor sets the sort index and order:

Function \_\_construct($index, $order = ‘ascending’) {

$this->\_index = $index;

$this->\_order = $order;

}

Function sort(array $list) {

//change the algorithm to match the sort preference:

If($this->\_order === ‘ascending’) {

Uasort($list, array($this, ‘ascSort’));

} else {

Uasort($list, array($this, ‘descSort’));

}

Return $list;

}

//functions that compares two values

Function ascSort($x, $y) {

Return strcasecmp($x[$this->\_index], $y[$this->\_index]);

}

Function descSort($x, $y) {

Return strcasecmp($y[$this->\_index], $x[$this->\_index]);

}

}

Class MultiNumberSort implements iSort {

//how to sort:

Private $\_order;

//sort index:

Private $\_index;

//constructor sets the sort index and order:

Function \_\_construct($index, $order = ‘ascending’) {

$this->\_index = $index;

$this->\_order = $order;

}

Function sort(array $list) {

//change the algorithm to match the sort preference:

If($this->\_order === ‘ascending’) {

Uasort($list, array($this, ‘ascSort’));

} else {

Uasort($list, array($this, ‘descSort’));

}

Return $list;

}

//functions that compares two values

Function ascSort($x, $y) {

Return ($x[$this->\_index] > $y[$this->\_index]);

}

Function descSort($x, $y) {

Return ($x[$this->\_index] < $y[$this->\_index]);

}

}

HTML

<?php

Require(‘iSort.php’);

Class StudentsList {

//stores the list of students:

Private $\_students = [];

Function \_\_construct($list) {

$this->\_students = $list;

}

//perform a sort using an iSort implementation:

Function sort(iSort $type) {

$this->\_students = $type->sort($this->\_students);

}

//display the students as an HTML list:

Function display() {

Echo ‘<ol>’;

Foreach($this->\_students as $student) {

Echo ‘<li>{$student[‘name’]} {$student[‘grade’]}</li>’;

}

Echo ‘</ol>’;

}

}

//create the array…

//array structure:

$students = [

256 => [

‘name’ => ‘Jon’,

‘grade’ => 98.5

],

2 => [

‘name’ => ‘Ali’,

‘grade’ => 85.1

]

];

//create the main object

$list = new StudentsList($students);

//show the original array:

Echo $list->display();

//sort by name:

$list->sort(new MultiAlphaSort(‘name’));

Echo $list->display;

//sort by grade:

$list->sort(new MultiNumberSort(‘grade’, ‘descending’));

$list->display();

//delete the object

Unset($list);

?>