

# Your First 5 Design Patterns

by Aaron Saray for MKEPUG

# What are They?

- Solving the Same Problem
  - But doing it right
  - System-i guys, Java (yuck), Cobol, they got it right
- Software Design has three parts
  - The "What" - business and functionality
  - The "How" - which design you choose
  - The "work" - implementation - or using 'how'
- Design Patterns how the HOW
  - They make the work a lot less sucky

# This isn't new

- Non PHP Programmers have seen this before
- PHP Guys - you've seen these before
  - PEAR DB
  - Zend Registry
  - Zend URI Factory
  - Doctrine's DAO
- Gang of four
  - **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**

# What they are NOT

- Not Plug and Play
- You can't just blindly copy
- Unproven theory
  - Design Laws?



Let's Do This

# #1 - Singleton Pattern

- Stop groaning
- The Singleton Design Pattern is used to restrict the number of times a specific object can be created to a single time by providing access to a shared instance of itself.
- Why might we use this?
  - Database
  - Heavy front build
  - Static

# #1 Continued

- Well 1 time isn't always true
  - Could restrict to 5 instances
- MMrrrr - wouldn't a proper registry of objects do away with this pattern?
  - Programmers aren't responsible

# Singleton Pattern Code Example

```
class InventoryConnection
{
    protected static $_instance = null;

    protected function __construct()
    {
        $this->_longWindedFunctionOne();
        $this->_delayedActionFunction();
    }

    public static function getInstance()
    {
        if (is_null(self::$_instance)) {
            self::$_instance = new self;
        }
        return self::$_instance;
    }
}

$inventory = InventoryConnection::getInstance();
```

# #2 Factory Pattern

- The Factory Design Pattern provides a simple interface to acquire a new instance of an object while sheltering the calling code from the steps to determine which base class is actually instantiated.
- Why might we use this?
  - View requesting content
  - Inventory system dealing with objects

# Factory Pattern Code

```
class Inventory
{
    public static function factory($type)
    {
        $class = 'Inventory' . ucwords($type);
        if (class_exists($class)) {
            return new $class;
        }
        else {
            throw new Exception("Type {$type} doesn't exist");
        }
    }
}

class InventoryDog {
    public function makeNoise(){
        echo "bark";
    }
}

class InventoryCat {
    public function makeNoise(){
        echo "mew";
    }
}

$currentAnimal = Inventory::factory('cat');
```

# #3 Observer Pattern

- The Observer Design Pattern facilitates the creation of objects that watch the state of a targeted functionality that is uncoupled from the core object.
- Why might we use this?
  - Plugins to software
  - Don't want to modify the shared core
  - Can't because of licensing
  - Enable / Disable auxiliary functionality without effecting base

# Observer Code

```
class Garden
{
    protected $_observers = array();

    public function attachObserver($type, $observer)
    {
        $this->_observers[$type][] = $observer;
    }
    protected function _notifyObservers($type)
    {
        if (!empty($this->_observers[$type])) {
            foreach ($this->_observers[$type] as $observer)
                $observer->notify($this);
        }
    }
    public function weed()
    {
        //action done here

        $this->_notifyObservers('afterweed');
    }
}
```

# Observer Code Continued

```
class GardenWeedObserverDance
{
    public function notify(Garden $garden)
    {
        echo 'Drop it like its hot because...';
        var_dump($garden);
    }
}

$garden = new Garden();
$dancer = new GardenWeedObserverDance();
$garden->attachObserver('afterweed', $dancer);
$garden->weed();
```

# #4 Decorator Pattern

- The Decorator Design Pattern is best suited for altering or decorating portions of an existing object's content or functionality without modifying the structure of the original object.
- Why might we use this?
  - quick small changes to internal content / values
  - modify user input (filter)
  - pretty output

# Decorator Code

```
class BadWord
{
    public $is = '.NET';
}

class BadWordDecorator
{
    public function decorate(BadWord $badword)
    {
        $badword->is = 'Microsoft ' . $badword->is;
    }
}

$badword = new BadWord();
$badWordDecorator = new BadWordDecorator();
echo $badword->is; // .NET
$badWordDecorator->decorate($badword);
echo $badword->is; // Microsoft .NET
```

# #5 - Strategy Pattern

- The Strategy Design Pattern helps architect an object that can make use of algorithms in other objects on demand in lieu of containing the logic itself
- Why might we use this?
  - reduce code duplication on similar models
  - no reinventing the wheel
  - quickly add a different process without changing the base object

# Strategy Code

```
class User
{
    protected $_imageStrategy;

    public function addImageStrategy($strategy)
    {
        $this->_imageStrategy = $strategy;
    }
    public function getThumbnail()
    {
        return $this->_imageStrategy->prepareThumbnail($this);
    }
}
class UserImage400pxStrategy
{
    public function prepareThumbnail(User $user)
    {
        //does some resizing stuff to 400px
        return $image;
    }
}

$user = new User;
$user->addImageStrategy(new UserImage400pxStrategy);
```

# What Next?

- Put this into practice
- Refactor
- Learn 12 more design patterns
  - how??

# Buy Professional PHP Design Patterns

On Amazon: <http://saray.me/bwK50V>

Also - one lucky winner tonite!

