PHP 5 and Design Patterns (DP)

Andrey Hristov
Dorten GmbH

Bulgarian Web Technologies Conference

March, 6th 2004

Who am I?

- Name : Andrey Hristov
- BSc in Computer Engineering
- Student at University of Applied Sciences, Stuttgart, Germany
- Programming for the Web since year 2000
- Working on the PHP core since year 2002
- Author of pecl/stats
- Currently working for Dorten GmbH, Germany

Survey

How many of you know something about Design Patterns?

How many of you know the new OO features of PHP 5?

PHP 5/ ZE 2 new features

- PPP modifiers (real encapsulation)
- Abstract methods and classes
- Iterators – native (SPL has prebuilt ones) and quite usable
- Interfaces (like in Java, but no interface vars)
- Class constants
- Static member variables
- Destructors
- Explicit object cloning
- Exceptions (mostly userland, only few extensions throw exceptions currently)
- Reflection API

PHP 5/ ZE 2 complex example

```php
<?php
interface ISome {
   public function func($p1);
}

interface IOther {
   public function func2($p1);
}

abstract class AClass
    implements ISome, IOther {
   const ACLASS_CONST = 1;
   private static $st = 0;

   function __construct() {}
   public function func($p1) {}
   public function func2($p2) {}
   protected abstract function afunc();
}
```

```php
class NAClass extends AClass {
   public function __construct(){
      parent::__construct();
   }
   public function __destruct(){echo "D\n";}
   private function __clone() {}
   protected function afunc() {}
   static function iterEx(ArrayObject $it) {
      foreach ($it as $k => $v)
         echo "$k -> $v\n";
      var_dump(get_class_methods($it));
      throw new Exception("Some exc");
   }
}

$a = array("one"=>1, "two"=> 2);
try {
  NAClass::iterEx(new ArrayObject($a));
} catch (Exception $e) {
  echo $e->getMessage()."\n";
  var_dump(get_class_methods($e));
}
?>
```

PHP 5 / ZE 2 complex example output

```
andrey@poohie:~> php php5_example.php
one -> 1
two -> 2
array(2) {
  [0]=>
  string(11) "__construct"
  [1]=>
  string(11) "getIterator"
}
Some exc
array(9) {
  [0]=>
  string(7) "__clone"
  [1]=>
  string(11) "__construct"
  [2]=>
  string(10) "getMessage"
  [3]=>
  string(7) "getCode"
  [4]=>
  string(7) "getFile"
  [5]=>
  string(7) "getLine"
  [6]=>
  string(8) "getTrace"
  [7]=>
  string(16) "getTraceAsString"
  [8]=>
  string(10) "__toString"
}
```

What are DP? (0)

- Simple and elegant, but sometimes tricky, solutions to specific types of problems in OO software design. A single pattern can apply to many different problems that are all with the same rough problems space.
- They do not require neither unusual language features nor amazing programming tricks but can fully utilize most of the OO features of a language.
- They lead to better system design. The benefits are most apparent later in development.

What are DP (1)

- Once you start to understand them, moving from a state of “Huh?” to “Ahaa!”, you will be changed forever.
- They give another level of abstraction and help when discussing with other developers
- First book on the matter :
  “Design Patterns – Elements of Reusable Object-Oriented Software”,
  by E. Gamma, R. Helm, R. Johnson, J. Vlissides
  (also known as GoF).

DP to be discussed

- Singleton
- Prototype
- Factory Method
- Abstract Factory
- Decorator
- Composite

Singleton (0) : Briefly

- One of the simplest and most known DP
- Insures that there will only one instance of a class and there is only one access point for accessing the instance
- One instance, when? DB classes, Classes which use "expensive" system resources. Have you used PHP + MySQL or PostgreSQL? If yes, then most likely you have used Singleton implicitly.
- The instance can be extended by subclassing, whenever needed, with no modification to the code that use it.

Singleton (1) : Example

```
<?php
class DB_SC {
   private static $sInst = NULL;
   private $dbConn  = NULL;

   protected function DB_SC()  { /* initialization */ }
   protected function __clone(){ /* nothing here   */ }
   public static function getInstance() {
      if (self::$sInst === NULL) {
         self::$sInst = new DB_SC();
      }
      return self::$sInst;
   }// getInstance
}// End of class DB_SC

$a = DB_SC::getInstance();
$b = DB_SC::getInstance();
var_dump($a, $b);
?>
```

Singleton (2) : Output

Output :

```
andrey@poohie:~> php singlet.php
object(DB_Singleton_Class)#1 (1) {
}
object(DB_Singleton_Class)#1 (1) {
}
```

Singleton (3) : Consequences

- Controlled access to the sole instance : since Singleton encapsulates the instance, it controls the access.
- Permits variable number of instances : more than one instance can be used and put into an array (object pool).
- Subclassing is possible : subclasses can be created for refining the operation. The code that use the singleton class requires very little change to start working with the new classes.
- Less polluted name space : less global variables sitting around (kicking out $GLOBALS).

Prototype (0) : Briefly

- The objects are created not by instantiating but by cloning : a prototype instance exists and whenever a new instance is needed then the prototype is cloned.
- Use “Prototype” when a system should be independent of how its objects are created and represented.
- When the instances of a class can have only few limited states then create objects for all states and later just clone the objects.

Prototype (1) : Example

```
<?php
class CT {
  const NONE     = 0;
  const PDA      = 1;
  const NOTEBOOK = 2;
  const DESKTOP  = 3;
  const CRAY     = 4;
  const MIN      = 0;
  const MAX      = 4;
}// End of class CT – Computer Type

class Computer {
  private static $pool = array();
  private static $count = array();
  private $type = CT::NONE;

  protected function Computer($t) {
    if ($t < CT::MIN || $t > CT::MAX){
      throw
       new Exception("Invalid type $t");
    }
    $this->type = $t;
  }

  function getPDA() {
    if (empty(self::$pool[CT::PDA])){
      self::$pool[CT::PDA] =
              new Computer(CT::PDA);
      self::$count[CT::PDA] = 0;
    }
    return clone self::$pool[CT::PDA];
  }

  function __destruct() {
    self::$count[$this->type]--;
  }

  protected function __clone() {
    self::$count[$this->type]++;
  }
}// End of class Computer

var_dump($a = Computer::getPDA());
var_dump($b = Computer::getPDA());

echo “Equal:”;var_dump($a === $b);
?>
```

Prototype (2) : Output

Output :

```
andrey@poohie:~> php proto.php
object(Computer)#2 (1) {
}
object(Computer)#3 (1) {
}
Equal:bool(false)
```

Prototype (3) : Consequences

- “Master” objects can be changed at run-time therefore giving more dynamic structure of the application.
- Reduced subclassing in some cases. See the example.
- In PHP 5 cloning is sometimes 3x faster than instantiation.

Factory Method (0) : Briefly

- The intent of Factory Method is that the base classes in the hierarchy define the interface for object creation but let the subclasses decide which class to instantiate (the base classes are abstract).
- Use it when the number of types (classes) of a system may increase significantly in the future. Factory Method will lower the quantity of the code to change.

Factory Method (1) : Example

```
<?php
class Circle extends Shape {
  protected function Circle() {
    echo "Circle created\n";
  }
  function draw() {}
}
class Rectangle extends Shape {
  protected function Rectangle() {
    echo "Rectangle created\n";
  }
  function draw() {}
}
class Triangle extends Shape {
  protected function Triangle() {
    echo “Triangle created\n";
  }
  function draw() {}
}
```

```
abstract class Shape {
   static function getInstance() {
      $params = func_get_args();
      $cName = $params[0];
      unset($params[0]);
      return new $cName();
   }
   abstract function draw();
}// End of class Shape

$a =Shape::getInstance("Circle");
$b =Shape::getInstance("Rectangle");

var_dump($a, $b);

$c = new Circle();

?>
```

Factory Method (2) : Output

```
andrey@poohie:~> php fact.php
Circle created
Rectangle created
object(Circle)#1 (0) {
}
object(Rectangle)#2 (0) {
}
PHP Fatal error:  Call to protected Circle::Circle()
from context '' in /home/andrey/test/web_conf/fact.php on line 37
```

Factory Method (3) : Consequences

- Factory Method provides hooks for subclasses : creating objects with factory method is more flexible than direct instantiation. The hook left by the base class is implemented by the subclasses and used by the base class.

Abstract Factory (0) : Briefly

- This Design Pattern provides an interface for creating families of related objects without specifying their concrete class names
- Helps ease the creation of portable applications.

Abstract Factory (1) : Example

```php
<?php
//Common interface
interface IInv {
   public function getGun();
   public function getRifle();
}
interface IGun{}
interface IRifle {}

//USSR - Red army
class AK47 implements IRifle { }
class Makarov implements IGun { }
class USSRInventory implements IInv {
   public function getGun() {
     return new Makarov();
   }
   public function getRifle() {
     return new AK47();
   }
}

//USA - US Army
class M16 implements IRifle{ }
class Beretta implements IGun{ }
class USInventory implements IInv {
   public function getGun() {
     return new Beretta();
   }
   public function getRifle() {
     return new M16();
   }
}

switch (@$argv[1]) {
  case 'USSR':
    $afact = new USSRInventory();
    break;
  case 'USA':
    $afact = new USInventory();
    break;
  default:
   die("Usage: php $PHP_SELF [name]\n".
        "name :=: USA | USSR\n");
}

var_dump($afact->getGun());

?>
```

Abstract Factory (2) : Output

```
andrey@poohie:~> php abstract_fact.php USSR
object(Makarov)#2 (0) {
}
andrey@poohie:~> php abstract_fact.php USA
object(Beretta)#2 (0) {
}
```

Abstract Factory (3) : Consequences

- Isolates concrete classes (their names). Abstract Factory helps you when creating instances of classes. The clients manipulate the classes through their abstract interfaces.
- Makes exchanging class “families” easy.
- Promotes consistency in the system.
- Adding new “horizontal” classes isn't that easy. Some interfaces must be extended and implemented.

Decorator (0) : Briefly

- The intent of this DP is attaching dynamically responsibilities to objects.
- Decorator provides good alternative to subclassing for behaviour extension.
- Decorator helps you preventing an explosion of classes in the hierarchy.
- Have you already experienced class explosion? Take a look at Decorator and maybe it will help you the make an implosion ;).
- Have you used Java? Java IO is based on Decorator.

Decorator (1) : Example

```
<?php
abstract class IIngr {
  private $nextIngr = NULL;
  protected $cost = 0.00;

  function IIngr(IIngr $i =NULL){
    $this->nextIngr = $i;
  }
  function getCost() {
    return $this->cost +
     ($this->nextIngr?
      $this->nextIngr->getCost():
      0
     );
   }
}
class Salami extends Iingr {
  protected $cost = 1.50;
}
class Ementaler extends IIngr {
  protected $cost = 1.80;
}
class Gouda extends IIngr {
  protected $cost = 1.80;
}
class FettaCheese extends IIngr {
  protected $cost = 0.50;
}
class Pizza extends IIngr {
  protected $cost = 0.25;
}

$p = new Pizza(
        new Salami(new Gouda())
     );
echo $p->getCost()."\n";
?>
```

Decorator (2) : Output

```
andrey@poohie:~> php decorator.php
3.55
```

Decorator (3) : Consequences

- More flexibility than static inheritance. It even can help you to escape the multiple inheritance nightmare.
- Avoids of over-engineered classes. All classes are simple and easy to “mix”. A new mixture is extremely easy to add.
- Easy to understand by the creators but sometimes hard to learn and debug by new team members.

Composite (0) : Briefly

- Composite lets you compose objects in tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions uniformly.
- Performing an operation on a node/composite also performs that operation on any children of that node/composite.

Composite (1) : Example

```
<?php
interface ICompanyPart {
  function getCost();
}

class Dep implements ICompanyPart {
  private $children = NULL;
  function Dep(ArrayObject $a){
    $this->children = $a;
  }
  function getCost() {
    $cost = 0;
    foreach($this->children as $v){
      $cost += $v->getCost();
    }
    return $cost;
  }
}
class Cwk implements ICompanyPart{
  private $salary = 0.00;
  function Cwk($salary) {
    $this->salary = $salary;
  }
  function getCost() {
    return $this->salary;
  }
}
$techDep=new Dep(new ArrayObject(
   array(new Cwk(200),
         new Cwk(250))));
$CEO = new Cwk(1000);
$supDep=new Dep(new ArrayObject(
   array(new Cwk(150),
         new Cwk(150))));
$company = new ArrayObject(
  array($techDep, $CEO, $supDep));
$a = new Dep($company);
echo $a->getCost()."\n";
?>
```

Composite (2) : Output

```
andrey@poohie:~> php composite.php
1750
```

Composite (3) : Consequences

- Helps when creating class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
- Allows the client to be simple.
- Makes it easier to add new components. They must obey to the interface and that's it.
- All this flexibility comes at a price : it's harder to restrict the components of the composite.

Questions?

I am reachable at :
andrey@php.net

The bulgarian PHP users mailing list :
general-bg@lists.php.net

This presentation is available at :
http://andrey.hristov.com/projects/php_stuff/pres/