# Classic Design Patterns in PHP

## Part One

LifeMichael
Haim Michael Blog

www.LifeMichael.com

abelski

www.abelski.com

## Design Pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

## Software Design Pattern?

“Software patterns are reusable solutions to recurring problems that we encounter during software development.”

Mark Grand, *Patterns in Java*. John Wiley & Sons, 2002.

## History

- The idea of using patterns evolves from the architecture field.

  Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977)

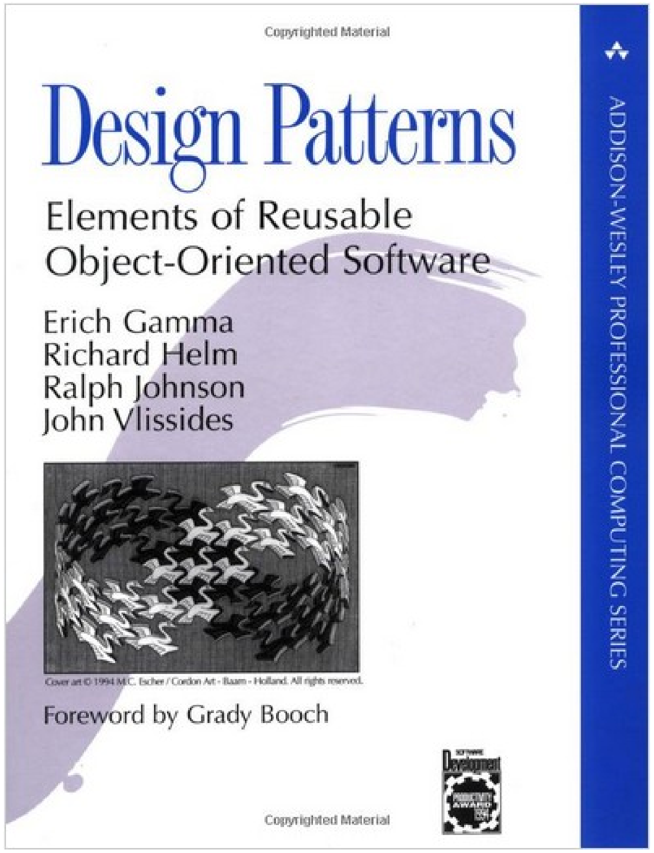
- The first GUI software patterns were set in 1987.

  Ward Cunningham and Kent Beck. *Using Pattern Languages for Object-Oriented Programs*. OOPSLA-87.

- The Gang of Four (AKA GOF) publish their “Design Patterns” book in 1994.

  Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Design Patterns.* Addison Wesley, 1994.

# History


Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

Copyrighted Material

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

This book defines 23 design patterns.

The design patterns are grouped into three categories:
Creational Patterns
Behavioral Patterns
Structural Patterns

## Why?

- Design patterns are language independent. Using design patterns we might improve our code.
- Design patterns define a vocabulary. The communication between developers within teams can improve.
- Design patters were already tested by other developers in other software systems.
- Using design patterns in general assist us with the development of a better software.

## Singleton

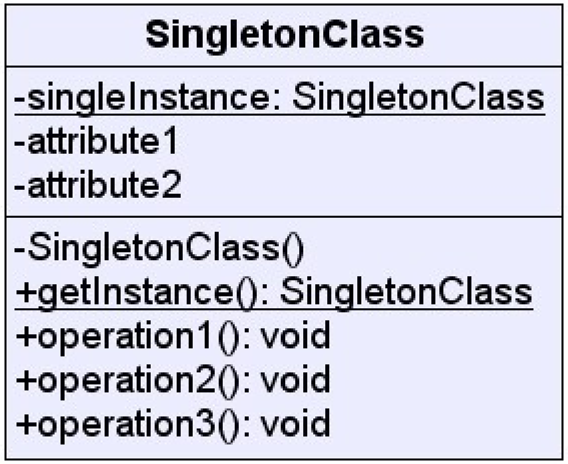
- **Problem Description**

  How to declare a class ensuring that it won't be possible to instantiate it more than once.

- **General Solution**

  We can declare a class with one (or more) private constructors only and include within that class a static method that once is called it checks whether an object of this class was already instantiated... if an object was already created then that static method will return its reference... if an object still wasn't created then it will instantiate this class, places the reference for the new object within a specific static variable and return it.

## Singleton


SingletonClass
-singleInstance: SingletonClass
-attribute1
-attribute2
-SingletonClass()
+getInstance(): SingletonClass
+operation1(): void
+operation2(): void
+operation3(): void

## Singleton

```
<?php
class Inventory
{
   private static $single;
   private function __construct() {  }
   public static function getInstance()
   {
      if(self::$single==null)
      {
         self::$single = new Inventory();
      }
      return Inventory::$single;
   }
}
?>
```

# Factory

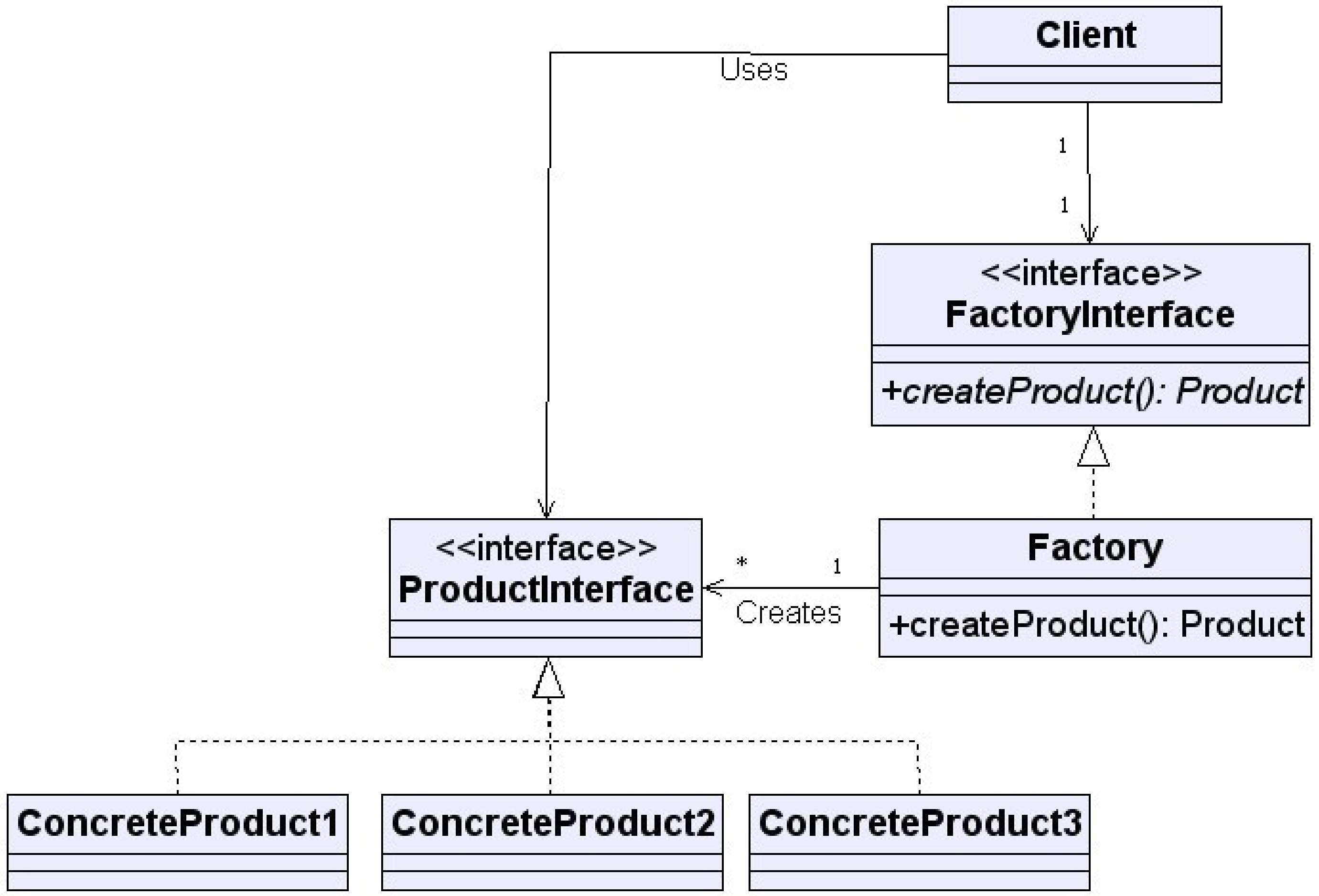
- Problem Description

  How to create an object of a type selected based on external data received in run time, without having the need to be aware of the actual type of the new created object.

- General Solution

  We will define an interface for creating new objects and let subclasses to decide about the exact class that should be instantiated according to the received arguments.

# Factory


Client
Uses
1
1
<<interface>>
FactoryInterface
+createProduct(): Product
<<interface>>
ProductInterface
*
1
Creates
Factory
+createProduct(): Product
ConcreteProduct1
ConcreteProduct2
ConcreteProduct3

## Factory

```php
<?php
class AnimalsFactory
{
    public static function getAnimal($str)
    {
        if($str=="cat")
        {
            return new Cat();
        }
        else if($str=="tiger")
        {
            $ob = new Cat();
            $ob->setCatType("tiger");
            return $ob;
        }
        else if($str=="dog")
        {
            return new Dog();
        }
        throw new Exception('animal type doesnot exist');
    }
}
```

# Factory

```
class Cat
{
    private $catType;
    function __toString()
    {
        return "cat " . $this->catType;
    }
    function setCatType($str)
    {
        $this->catType = $str;
    }
}

class Dog
{
    function __toString()
    {
        return "dog";
    }
}
?>
```

# Proxy

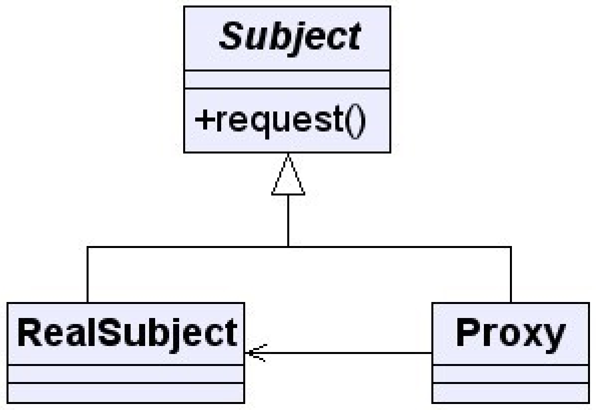
- **Problem Description**

  How to ensure that a costly object will be instantiated on demand (only when we actually need it).

- **General Solution**

  Define a class that its objects will be able to maintain a reference for the costly object and will be capable of creating it (on demand... when there is a need for it).

# Proxy



## Proxy

```
<?php
interface IVideo
{
    function play();
}

class Video implements IVideo
{
    private $title;
    public function __construct($title)
    {
        $this->title = $title;
    }
    public function play()
    {
        echo "playing ".$this->title." video";
    }
}
```

## Proxy

```
class VideoProxy implements IVideo
{
    private $video;
    private $title;
    public function __construct($str)
    {
        $this->title = $str;
    }
    public function play()
    {
        if($this->video==null)
            $this->video = new Video($this->title);
        $this->video->play();
    }

}

$ob = new VideoProxy("XMania III");
$ob->play();
?>
```

## Template

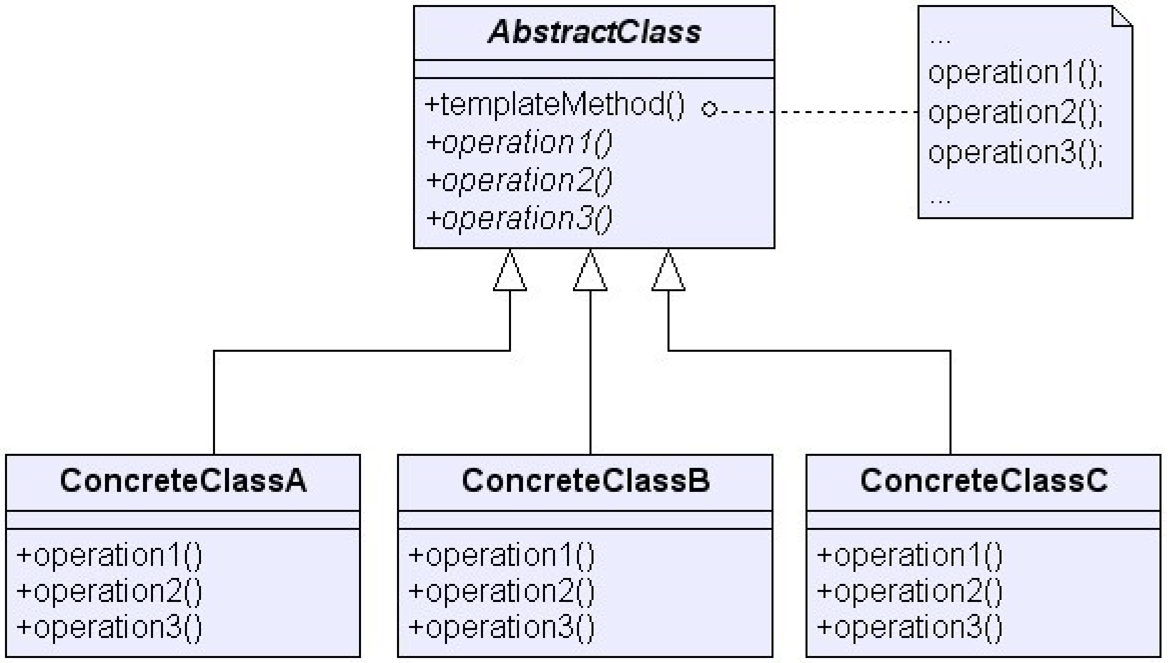
- **Problem Description**

  How to define a skeleton of algorithm allowing to defer some of its steps to the subclasses... ?

- **General Solution**

  We will define an abstract class with a concrete method that implements the algorithm by calling abstract methods declared in the same class. Classes that extend our abstract class will include their implementations for these abstract methods.

## Template



*AbstractClass*
+templateMethod()
*+operation1()*
*+operation2()*
*+operation3()*
...
operation1();
operation2();
operation3();
...
ConcreteClassA
+operation1()
+operation2()
+operation3()
ConcreteClassB
+operation1()
+operation2()
+operation3()
ConcreteClassC
+operation1()
+operation2()
+operation3()

## Template

```
<?php

abstract class Shape
{
    public abstract function area();
    public function __toString()
    {
        return "my area is ".$this->area();
    }
}
```

## Template

```
class Circle extends Shape
{
    private $radius;
    public function __construct($rad)
    {
        $this->setRadius($rad);
    }
    public function getRadius()
    {
        return $this->radius;
    }
    public function setRadius($radius)
    {
        $this->radius = $radius;
    }
    public function area()
    {
        return pi()*$this->getRadius()*$this->getRadius();
    }
}
```

## Template

```
class Rectangle extends Shape
{
    private $width;
    private $height;
    public function __construct($wVal, $hVal)
    {
        $this->setWidth($wVal);
        $this->setHeight($hVal);
    }
    public function getWidth()
    {
        return $this->width;
    }
    public function setWidth($width)
    {
        $this->width = $width;
    }
    public function getHeight()
    {
        return $this->height;
    }
```

## Template

```
    public function setHeight($height)
    {
        $this->height = $height;
    }
    public function area()
    {
        return $this->getWidth()*$this->getHeight();
    }
}

$vec = array(new Rectangle(3,4), new Circle(5), new Rectangle(5,4));
foreach($vec as $ob)
{
    echo "<br>".$ob;
}

?>
```

## Decorator

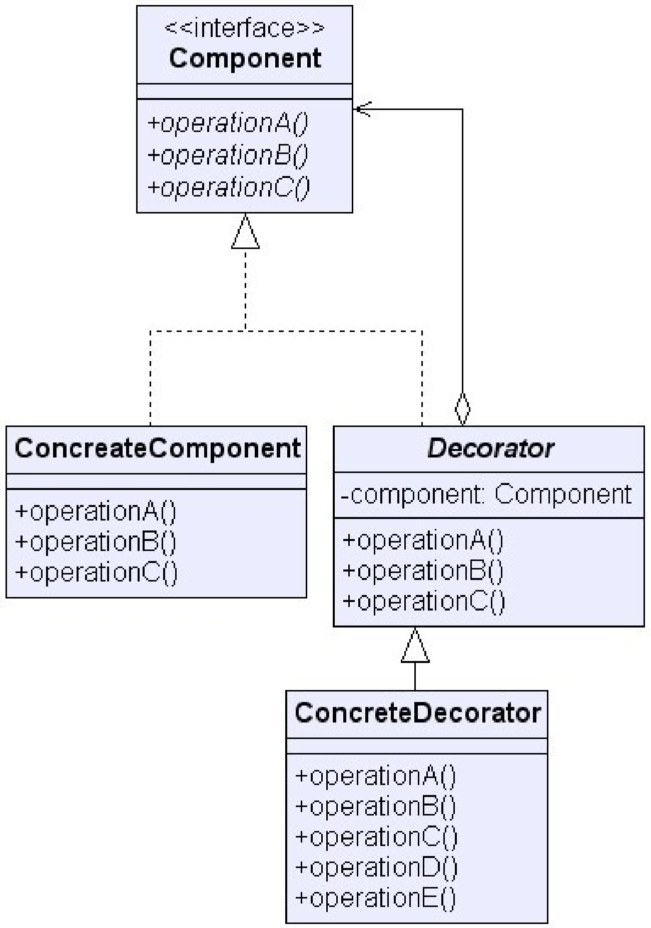
- Problem Description

  How to extend the functionality of a given object in a way that is transparent for its clients without sub classing the given object.

- General Solution

  Declaring a new class, that implements the same interface the original object implements and delegates operations to the original given object.

# Decorator



<<interface>>
Component
+operationA()
+operationB()
+operationC()
ConcreateComponent
+operationA()
+operationB()
+operationC()
Decorator
-component: Component
+operationA()
+operationB()
+operationC()
ConcreteDecorator
+operationA()
+operationB()
+operationC()
+operationD()
+operationE()

# Decorator

```
<?php
interface IBook
{
    function setTitle($str);
    function getTitle();
    function setPages($num);
    function getPages();
    function setAuthor($author);
    function getAuthor();
}
```

## Decorator

```
final class StandardBook implements IBook
{
   private $author;
   private $title;
   private $pages;

   public function getAuthor()
   {
      return $this->author;
   }

   public function getPages()
   {
      return $this->pages;
   }
```

## Decorator

```
    public function getTitle()
    {
        return $this->title;
    }

    public function setAuthor($author)
    {
        $this->author = $author;
    }

    public function setPages($pages)
    {
        $this->pages = $pages;
    }

    public function setTitle($title)
    {
        $this->title = $title;
    }
}

interface ILibraryBook extends IBook
{
    function printDetails();
}
```

# Decorator

```
class LibraryBook implements ILibraryBook
{
   private $book;

   public function __construct()
   {
      $this->book = new StandardBook();
   }

   public function printDetails()
   {
      echo "<br>author=".$this->book->getAuthor();
      echo "<br>pages=".$this->book->getPages();
      echo "<br>title=".$this->book->getTitle();
   }
```

## Decorator

```
public function getAuthor()
{
    return $this->book->getAuthor();
}

public function getPages()
{
    return $this->book->getPages();
}

public function getTitle()
{
    return $this->book->getTitle();
}

public function setAuthor($str)
{
    $this->book->setAuthor($str);
}
```

## Decorator

```
        public function setPages($num)
        {
            $this->book->setPages($num);
        }

        public function setTitle($str)
        {
            $this->book->setTitle($str);
        }
    }

    $book = new LibraryBook();
    $book->setAuthor("haim");
    $book->setPages(320);
    $book->setTitle("core scala");
    $book->printDetails();

    ?>
```

## Prototype

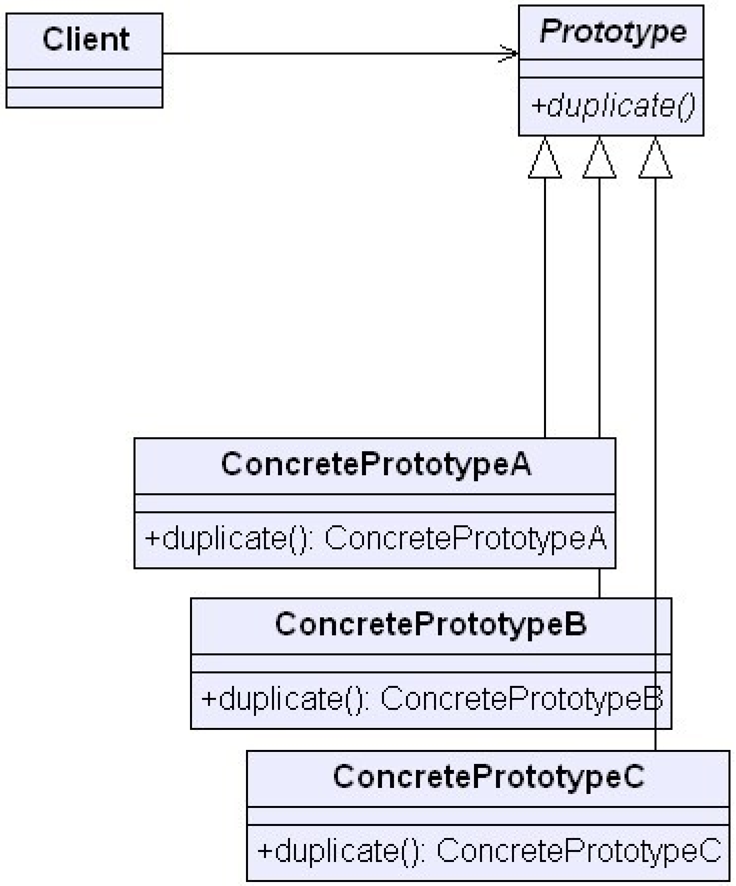
- **Problem Description**

  How to create new objects based on a prototype object we already have, without knowing their exact class or the details of how to create them.

- **General Solution**

  We will declare an abstract class that includes the clone() method. When calling the clone() method a new object is instantiated from the same class the object (on which the clone() method is called) was instantiated from.

# Prototype


Client
Prototype
+duplicate()
ConcretePrototypeA
+duplicate(): ConcretePrototypeA
ConcretePrototypeB
+duplicate(): ConcretePrototypeB
ConcretePrototypeC
+duplicate(): ConcretePrototypeC

## Prototype

```
<?php
interface Duplicatable
{
    function duplicate();
}

class Circle implements Duplicatable
{
    private $radius;
    public function __construct($rad)
    {
        $this->setRadius($rad);
    }
    public function getRadius()
    {
        return $this->radius;
    }
    public function setRadius($radius)
    {
        $this->radius = $radius;
    }
```

## Prototype

```
    public function duplicate()
    {
        return new Circle($this->getRadius());
    }
    public function area()
    {
        return pi()*$this->getRadius()*$this->getRadius();
    }
    public function __toString()
    {
        return "i m a circle... my area is ".$this->area();
    }
}
```

## Prototype

```
class Rectangle implements Duplicatable
{
    private $width;
    private $height;
    public function __construct($wVal, $hVal)
    {
        $this->setWidth($wVal);
        $this->setHeight($hVal);
    }
    public function getWidth()
    {
        return $this->width;
    }
    public function setWidth($width)
    {
        $this->width = $width;
    }
    public function getHeight()
    {
        return $this->height;
    }
```

## Prototype

```
    public function setHeight($height)
    {
        $this->height = $height;
    }
    public function duplicate()
    {
        return new Rectangle($this->getWidth(),$this->getHeight());
    }
    public function area()
    {
        return $this->getWidth()*$this->getHeight();
    }
    public function __toString()
    {
        return "i m a rectangle... my area is ".$this->area();
    }
}

$obA = new Circle(3);
$obB = $obA->duplicate();
echo $obB;

?>
```