OBJECT ORIENTED & DESIGN PATTERNS - PHP5

By Ziv Perry
http://fat-fish.co.il

zend

The PHP Company

Copyright © 2007, Zend Technologies Inc.

תכנות מוכוון עצמים

zend

The PHP Company

תכנות מבני

מערכות תכנה הינן מסובכות מטבען, אלמנטים לסיבוך:

- סיבוך במרחב הבעיה.
- הקושי בתהליך הפיתוח.
- מספר גדול של שורות קוד – קושי בהבנה.
- דרישות משתנות במהלך הפיתוח.

כדי להתגבר על אלמנטי הסיבוך משתמשים ב- "decomposition".

מחלקים את המערכת לחלקים יותר ויותר קטנים. כל חלק ניתן לשליטה והבנה קלה יותר.

הגישה המסורתית – פירוק אלגוריתמי

- הגישה לפירוק הנה לפי האלגוריתם.
- כל מודול במערכת מביע צעד עיקרי בתהליך הכולל.
- חלוקת המערכת לאלמנטים פונקציונלים.
- תת-התוכניות מהוות את אבני הבניה.
- שבירת הסיבוכיות על ידי פירוק בעיה גדולה לצעדים קטנים יותר.

גישת העצמים – פירוק מוכוון עצמים

תכנות מוכוון עצמים – היא פרדיגמת תכנות המשתמשת בעצמים לשם תכנות תכניות מחשב.

- **המקור לאובייקטים נבע מהקשר הצמוד בין מבנה נתונים מסויים לפונצקיות הפועלות עליו בתכנות הפרוצדוראלי. שינוי במבנה הנתונים, גרר שינוי בפונקציות המטפלות בנתונים אלו וגרם לבעיות תחזוקה.**
- **על מנת לאחד בין שני הגורמים – נוצרו האובייקטים, יחידות עצמאיות של נתונים המכילים גם פונקציונליות. את האובייקטים ניתן לחלק לקטגוריות שונות ולקיים תלויות ויחסים שונים בינהם, להעניק לכל אובייקט מאפיינים ושיטות הפועלות עליו.**

גישת העצמים – פירוק מוכוון עצמים (המשך...)

- **אנו מזהים את האובייקטים של הבעיה, הנגזרים ישירות מאוצר המלים של הבעיה. כלומר, נשתמש בשפה "טבעית" לתאר את העצמים שאנו מזהים.**
  - מגדירים את התכונות של האובייקטים.
  - מגדירים את הפעולות (Methods) של האובייקטים.
- **"העולם" – קבוצת גופים אוטונומיים משתפי פעולה לביצוע התנהגות ברמת תיאור גבוהה יותר. פתרון הבעיה הינו סדרת קריאות לפעולות של האובייקטים.**
- **החלוקה מבוססת על אובייקטים ולא על אלגוריתם ונקראת לכן – Object Oriented Decomposition.**
  - פירוק אלגוריתמי – מציין את סדר האירועים.
  - פירוק OO – מדגיש את הישויות הפועלות.

יתרונות פירוק מוכוון עצמים

- שימוש חוזר (re-use) של גורמים משותפים.
- קל לשינויים ולהתפתחות.
- עוזר להתגבר על הסיבוכיות האינהרנטית של התוכנה בעזרת בחירה נבונה של הפרדת נושאים במרחב מצבים גדול.
- אבסטרקציה (הפשטה).

אבסטרקציה

- **הפרט יכול להבין, לעבד ולזכור כמות מוגבלת של אינפורמציה. מתגברים על בעיה זו באמצעות הפרדת האירועים למימדים אחדים ושבירת רצף האינפורמציה לקבוצות.**
- **אבסטרקציה הנה טכניקה המתמודדת עם הסיבוכיות על-ידי התעלמות מפרטים קטנים ומבט על מודל מוכלל של אובייקט. התרכזות בפרט אחד בכל פעם – ולא בכל הפרטים ביחד.**
- **גישת הפירוק מוכוון העצמים:**
  - עצמים הינם אבסטרקציות (הפשטה) של גופים בעולם האמיתי, המייצגים בתוכם פרטי אינפורמציה רבים.

אובייקט

- **אובייקט הינו כל רכיב שנוכל לאפיינו. בעולם האמיתי, אובייקטים יכולים להיות: שולחן, מכונית, מחשב וכו...**
- **כל אובייקט נגדיר על ידי שלושה מאפיינים:**
  - זהות (id) - שם האובייקט
  - מצב (state) - סך התכונות הסטטיות והדינמיות של של האובייקט.
  - התנהגות (behavior) - סך הפעולות בהם ניתן לשנות את המצב.

אובייקט (המשך...)

- **דוגמה למאפיינים של אובייקט:**
  - זהות: מכונית
  - מצב: דגם, נפח מנוע, מהירות, מספר גלגלים, הילוך נוכחי...
  - התנהגות: האצה, האטה, התנעה...

במימוש אובייקט, משתנים מייצגים את המצב ושגרות (פונקציות/מתודות) את ההתנהגות, כאשר דרך שגרות אלו, אנו משנים את מצב האובייקט.

אובייקט (המשך...)

- **דוגמה נוספת:**
  - זהות: רשימה (israel, palestine, bulgaria, china...)
  - מצב: האלמנטים של הרשימה, מספר האלמנטים ברשימה...
  - התנהגות: הוספת אלמנט, מחיקת אלמנט, מיון...

באופן זה, אנו יכולים לומר כי כל מה שהאובייקט יודע ויכול לעשות, מבוטא על ידי המשתנים והשגרות של האובייקט.

עקרונות תכנות מוכוון עצמים

- ***Encapsulation*** **-** עקרון ההסתרה. כלומר, האובייקט הינו "קופסה שחורה" כלפי חוץ. התהליכים הקורים בתוך האובייקט אינם ידועים כלפי חוץ ואין צורך בידיעתם. העקרון תורם למודולריות ופשטות.

עקרונות תכנות מוכוון עצמים (המשך...)

- ***Inheritance*** – מחלקה ניתן להגדיר על בסיס מחלקה קיימת, ובכך להרחיב את יכולות המחלקה "הנורשת" ללא צורך לשכפל קוד. הרחבת מחלקה (מחלקת האב, או מחלקת הבסיס) כוללת הוספת properties ו-methods למחלקה הנגזרת.

עקרונות תכנות מוכוון עצמים (המשך...)

- ***Polymorphism*** – מבוסס על תכונת האבסטרקציה של OO. עקרון זה, מעניק את הכח לאפשר התנהגויות שונות או תכונות שונות לאובייקטים שונים – אך בעלי ממשק זהה. כלומר, אובייקטים שמספקים את אותה "מראה" כלפי חוץ – יתנהגו בצורה שונה.
תכונות הנדרשות לצורך polymorphism הינם העמסה ודריסה (overloading and overriding).

מחלקה

- המרכיב הבסיסי ב-Object Oriented Programming הינו המחלקה – class. מחלקה הינה מבנה נתונים מופשט, בעל תכונות (properties) ופעולות (methods) המייצגות את המצב וההתנהגות של האובייקט הנוצר ממחלקה זו.
- אובייקט הינו מופע (instance) או ישות של המחלקה, כלומר המחלקה היא רק ייצוג של האובייקט ולא האובייקט עצמו.
  - אם נחזור לדוגמת המכונית, אזי המחלקה היא ייצוג כללי ומופשט של מכונית – בעוד האובייקט מייצג מכונית ספציפית עם נתונים מסויימים.
- מכאן ניתן לומר כי כל האובייקטים של המחלקה, הינם בעלי אותה התנהגות ובעלי אותם משתני מצב – אך הערכים של אותם משתנים הינם שונים.

מחלקה (המשך...)

- **ישנן שתי דרכים להבחין במחלקה:**
  - ממשק (interface) – המראה החיצוני של המחלקה המדגיש את האבסטרקציה ומסתיר את פרטי המימוש.
  - מימוש (implementation) – פרטי מימוש המחלקה ורכיבי המידע שהיא מכילה.
- **יתרונות:**
  - מודולריות והסתרת מידע.
  - שימוש חוזר (re-use).

אובייקטים ב-PHP5

zend

The PHP Company

הגדרת מחלקה ויצירת אובייקט

- class – הגדרת מחלקה.
- new – יצרת מופע של אובייקט ממחלקה מסויימת.
- this - ייצוג של האובייקט הנוכחי מתוך טווח ההכרה של האובייקט.

```
class Foo {

}
$obj = new Foo();
var_dump($obj);
```

```
object(Foo)#1 (0) {
}
```

פונקציות

- שיטות (methods) הינן פונקציות המוגדרות באובייקט – או הפעולות המוגדרות על האובייקט.
- ניתן לקרא לפונקציה של האובייקט על ידי שימוש ב '->'
- ניתן לקרא בצורה סטטית לפונקציות של המחלקה על ידי שימוש ב '::'

```
class Foo {

    function bar() {
        echo "Foo::bar";
    }

    function baz() {
        $this->bar();
    }
}

$a = new Foo();
$a->baz();
Foo::bar();
```

```
Foo::bar
Foo::bar
```

משתנים

- משתנים של אובייקט (data members), הינם המאפיינים של האובייקט.
- פנייה אל המשתנים הינה זהה לזו של פנייה לפונקציה.

```
class Foo {

    public $var;

    public function bar($int) {
        $this->var = $int;
    }
}

$a = new Foo();
$a->bar(2);
echo $a->var;
```

```
2
```

העברה והעתקה של אובייקטים

- **אובייקטים ב php5 מיוצגים תמיד על ידי ייחוס (reference).**
- **ליצירת העתק של אובייקט נשתמש באופרטור clone.**

```
class Foo {
    public $var;
}

$a = new Foo();
$a->var = 2;
$b = $a;
echo $b->var;
```

```
2
```

```
function bar($obj) {
    $obj->var = 3;
}

bar($a);
echo $a->var;
```

```
3
```

```
$b = clone $a;
$b->var = 4;
echo $a->var;
echo $b->var;
```

```
3
4
```

בנאי

- constructor הינה פונקציה מיוחדת של המחלקה. הנקראת באופן אוטומטי עם יצירת מופע שלה.
  - אנו משתמשים בפונקציה זו על מנת לאתחל את האובייקט.
  - בדרך כלל לא נהוג לבצע פעולות בבנאי למעט איתחול – אם כי עקרונית, אין מניעה לעשות זאת.
- פונקציות ה constructor אינה מחזירה ערך! לכן – כישלון של פעולה בפונקציה זו מבוטא בדרך כלל על ידי חריגה (exception).

```
class Foo {

    public $var;

    public function __construct() {
        $this->var = 2;
    }
}
```

מפרק

- **destructor הינה פונקציה מיוחדת של המחלקה. הנקראת באופן אוטומטי עם שיחרור האובייקט.** ***(מתי משתחרר אובייקט מהזיכרון?)***
  - פונקציה זו נועדה לשחרר משאבים שנוצרו על ידי האובייקט.

```
class Foo {

        public $file;

        public function __construct() {
                $this->file = fopen('/tmp/file');
        }

        public function __destruct() {
                fclose($this->file);
        }
}
```

הרשאות גישה

- public – גישה לפונקציה/משתנה מכל מקום, הן מחוץ לאובייקט והן בתוכו, גם במקרה של הורשה.

```
class Foo {

    public $var;

    public function func() {
        return $this->var;
    }
}
$a = new Foo();
$a->var;
$a->func();
```

הרשאות גישה (המשך...)

- **private – גישה לפונקציה/משתנה רק בתוך תחום האובייקט. אין גישה לאלמנט פרטי ממחלקות יורשות.**

```
class Foo {

    private $var;

    private function _func() {
        return $this->var;
    }
    public function func() {
        return $this->_func();
    }
}
$a = new Foo();
$a->func();
$a->_func();
$a->var;
```

```
Fatal error:  Call to private
 method Foo::_func() from context

Fatal error:  Cannot access
 private property Foo::$var
```

הרשאות גישה (המשך...)

- **protected – זהה להרשאה private עם שינוי קל, הרשאה זו זמינה עבור מחלקות יורשות.**

```
class Foo {

    protected $var;

    protected function _func() {
        ...
    }
}
```

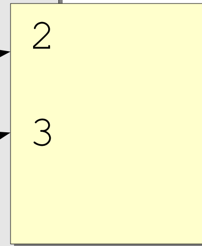
גישה סטטית

- במחלקה ניתן להגדיר משתנים ופונקציות סטטיות על ידי שימוש במילת המפתח static.
- פונקציות ומשתנים סטטיים הינם שייכים למחלקה ולא לאובייקט. והגישה אליהם נעשית באמצעות האופרטור :: ובעזרת שם המחלקה או מילת המפתח self מתוך הגדרת המחלקה.
- ניתן להשתמש בשתנים סטטיים לשיתוף מידע בין כל האובייקטים הנוצרים מאותה מחלקה.
- מילת המפתח this (המייצגת את האובייקט הנוכחי) אינה זמינה בפונקציות סטטיות.

גישה סטטית (המשך...)

- דוגמה לשימוש בקריאות סטטיות ושיתוף מידע בין אובייקטים.

```
class Foo {

    private static $counter = 0;

    public function __construct() {
        ++self::$counter;
    }

    public function getCount() {
        return Foo::$counter;
    }

    public static function getStaticCount() {
        return self::$counter;
    }
}
$a = new Foo();
$b = new Foo();
echo $a->getCount();
$c = new Foo();
echo Foo::getStaticCount();
```



קבועים

- קבועים (constants) של מחלקות הינם זהים בהתנהגותם למשתנים סטטיים למעט העובדה שחובה לאתחל אותם, הם לא ניתנים לשינוי והם יכולים להכיל רק סקלרים.

```
class Foo {

    const CONSTANT = 1;

    const STR = 'string';

    public function bar() {
        echo self::STR;
    }
}
$a = new Foo();
$a->bar();
echo Foo::CONSTANT;
```

```
1

string
```

ירושה

- מחלקה יכולה לרשת מחלקה אחת בלבד.
- ירושה מקיימת את יחס is_a בין מחלקות.
- בדוגמאות הבאות נבחן את עקרונות הירושה:

```
class A {

    public function foo() {}
}

class B extends A {

}

$b = new B();
$b->foo();
```

```
B is_a A
```

ירושה (המשך...)

- הרשאות גישה בירושה:

```
class A {

    protected $var1;

    private $var2;

    protected function foo() {}

    private function bar() {}
}

class B extends A {

    public function func() {
        $this->foo();
        $this->bar;

        return $this->var1;
    }
}
```

- מחלקה B נראית כעת כך:

```
class B {

    protected $var1;

    protected function foo() {}

    public function func() {}
}
```

ירושה (המשך...)

```
class A {

    protected $var1 = 1;

}

class B {

    protected $var1 = 2;

    public function func() {
        return $this->var1;
    }
}

$b = new B();
echo $b->func();
```

- **בירושה, ניתן לדרוס פונקציות ומשתנים.**

ירושה (המשך...)

- נשתמש במילת המפתח parent על מנת לגשת לפונקציות או משתנים השייכים למחלקת האב (המחלקה הנורשת).
- parent דומה למילת המפתח לגישה סטטית self, והיא משמשת הן לגישה סטטית והן לגישה לאובייקט – על פי האלמנט אליו היא מצביעה.

```php
class A {

    public function foo() {
        echo 'A';
    }
}

class B extends A {

    public function foo() {
        parent::foo();
        echo 'B';
    }
}
$a = new A();
$a->foo();
$b = new B();
$b->foo();
```

```
A
AB
```

ירושה (המשך...)

- ניתן למנוע ירושת מחלקה על ידי הגדרתה כ-final.

```
final class A { }

class B extends A { }

$b = new B();
```

```
Fatal error: Class B may not
 inherit from final class (A)
```

- ובאותה מידה, ניתן למנוע ממחלקה יורשת לממש (לדרוס) פונקציה הקיימת במחלקה הנורשת על ידי שימוש במילה final.

```
class A {

    final public function foo() {}
}

class B extends A {
    public function foo() {}
}

$b = new B();
```

```
Fatal error: Cannot override
 final method A::foo()
```

אופרטורים של אובייקטים ומחלקות

- הפונקציה is_a בודקת אם מחלקה מסויימת הינה מסוג מסויים או מקיימת יחס ירושה (is_a).

```
class A {}

class B extends A {}

$a = new A();
$b = new B();

var_dump($a, 'A');
var_dump($a, 'B');
var_dump($b, 'A');
var_dump($b, 'B');
```

- php5 מספקת את האופרטור instanceof שאינו מצריך שימוש בשם המחלקה כמחרוזת.

```
var_dump($a instanceof A);
var_dump($b instanceof B);
```

- פונקציות נוספות הפועלות על מחלקות ואובייקטים:
http://php.net/manual/en/ref.classobj.php

מחלקה אבסטרקטית

- מחלקה אבסטרקטית הינה מחלקה שלא ניתן ליצור ממנה אובייקטים.
- מחלקה זו היא למעשה תבנית בלבד שניתן אך ורק לרשת ממנה.
- מחלקה המכילה פונקציה אבסטרקטית אחת לפחות הינה בהכרח מחלקה אבסטרקטית, אך ניתן ליצור מחלקה אבסטרקטית שאין בה פונקציות אבסטרקטיות.

```
abstract class A {

    abstract public function foo();

    public function bar() {
        ...
    }
}

class B extends A {

    public function foo() {
        ...
    }
}
```

מחלקה אבסטרקטית (המשך...)

- פונקציה אבסטרקטית הינה פונקציה שאנו מציינים רק את אב הטיפוס (חתימה) שלה ללא מימוש ומשאירים את המימוש שלה למחלקה היורשת.
- מימוש פונקציה אבסטרקטית מחייב שמירה על הרשאת גישה זהה או פחותה.

```
abstract class A {

    abstract protected function foo();
    abstract public function baz();
}

class B extends A {

    public function foo() {...}
}
```

```
Fatal error: Class B contains 1 abstract method and must
 therefore be declared abstract or implement the remaining
 methods (A::baz)
```

ממשק

- מחלקה אבסטרקטית מהווה תבנית לירושה המספקת אפשרות למימוש חלקי. ממשק (interface) הינה הגדרת אבטיפוס בלבד.
- ממשק מייצג רק את ההחצנה של האובייקט (אבסטרקציה) שלו ולא מציין כלום לגבי המימוש.
- הממשק מציין רק את הפעולות שחושף האובייקט כלפי חוץ, כלומר את הפונקציות הציבוריות שלו.
- אובייקט יכול לרשת רק מחלקה אחת, אך אינו מוגבל בממשקים שהוא יכול לממש.

```
interface A {

    public function foo();

    public function bar();
}
```

מחלקות ואובייקטים - דוגמאות

- התבקשנו ליצור מחלקה המייצגת מלבן עבור אפליקציה גאומטרית:

```
class Rectangle {
    private $w;
    private $h;

    public function setDiamensions($h, $w) {
        $this->w = $w;
        $this->h = $h;
    }

    public function getArea() {
        return $this->w *
               $this->h;
    }

    public function getPerimeter() {
        return 2*($this->w + $this->h);
    }
}
```

מחלקות ואובייקטים – דוגמאות (המשך...)

- להרחבת האפליקציה, אנו זקוקים למחלקה נוספת, מעגל:

```
class Circle {

    private $r;
    const PI = 3.14;

    public function getArea() {
        return Circle::PI *
               $this->r *
               $this->r;
    }

    public function getPerimeter() {
        return 2 *
               Circle::PI *
               $this->r;
    }

    public function setRadius($r) {
        $this->r = $r;
    }
}
```

מחלקות ואובייקטים – דוגמאות (המשך...)

- פונקציה מסויימת באפליקציה, ידעה לקבל מלבן – ולהחזיר את השטח שלו. כעת אנו רוצים שהפונקציה תהיה כללית יותר ותעבוד גם עבור המעגל.

```
function get_shape_area($rect) {
    if ($rect instanceof Rectangle) {
        return $rect->getArea();
    } else {
        // error
    }
}
```

- מהתבוננות במעגל ובמלבן אנו למדים שהם חולקים פונקציות זהות. נוכל להכריז על פונקציות אלו כממשק של מעגל ומלבן (ואולי כל צורה?)

```
interface Shape {

    public function getArea();

    public function getPerimeter();
}
```

מחלקות ואובייקטים – דוגמאות (המשך...)

- כעת כל מחלקה שאנו רוצים שתתנהג כ"צורה" – נממש אותה על פי הממשק Shape.

```
class Rectangle implements Shape {
    ...
}
class Circle implements Shape {
    ...
}
```

- כעת, הפונקציה תמיד תצפה לקבל אובייקט שעומד ב"חוזה" של Shape.

```
function get_shape_area($rect) {
    if ($rect instanceof Shape) {
        return $rect->getArea();
    } else {
        // error
    }
}
```

מחלקות ואובייקטים – דוגמאות (המשך...)

- **על מנת לחסוך את תהליך הבדיקה של האובייקט בתוך הפונקציה, php5 מספקת לנו type hinting.**

```
function get_shape_area(Shape $rect) {
    return $rect->getArea();
}
```

- **אם אנו מעבירים פרמטר שלא מתאים למחלקה שהוגדרה (יחס is_a) אזי אנו מקבלים שגיאה מסוג:**

```
Catchable fatal error: CArgument 1 passed to function() must
 be an instance of Interface/Class, Somthing given
```

- **type hinting עובד עבור מחלקות, ממשקים ומערכים בלבד.**

מחלקות ואובייקטים – דוגמאות (המשך...)

- לאור התרחבות הדרישות של האפליקציה, נדרש מאיתנו לספק נתון צבע עבור כל צורה.
- בתחילה הוספנו את המשתנה color לכל מחלקה, וכן פונקציות להשמה וקבלה של צבע זה.
- האם יש צורך להוסיף את הפונקציה והמשתנה עבור כל מחלקה חדשה של צורה שניצור?

```
class Rectangle {

    private $color;

    public function getColor() {
        return $this->color;
    }
    ...
}
```

מחלקות ואובייקטים – דוגמאות (המשך...)

- **לפתרון הבעיה, נחליף את הממשק Shape במחלקה אבסטרקטית בשם זה, המספקת לנו את התשתית שאנו רוצים שתהיה זהה בכל ה"צורות" שלנו. ונחליף את ה"מימוש" של מחלקה זו על ידי הצורות בהורשה.**
- **ניתן כמובן לשמור על הממשק הישן ולהוסיף את התכונות החדשות למחלקה אבסטרקטית שתממש את Shape ואותה יירשו הצורות.**

```
abstract class Shape {

    abstract function getArea();
    abstract function getPerimeter();

    protected $color;

    public function getColor() {
        return $this->color;
    }
    ...
}

class Rectangle extends Shape {...}
```

פונקציות קסומות

- Php5 מספקת מספר פונקציות "קסומות" (magic functions) לעבודה עם אובייקטים.
- כל הפונקציות מתחילות עם __ (שני קווים תחתונים) בדיוק כמו הבנאי והמפרק, והן:
  - call, get, set, isset, unset, sleep, wakeup, toString, set_state ו-clone.

sleep ו-wakeup

- כאשר אנו מבצעים סריאליזציה של אובייקט, אנו מקבלים מחרוזת שמתארת את האובייקט לצורך יצירתו מחדש.

```
class Foo {
    private $a;
    protected $b;

    public function set($a, $b) {
        $this->a = $a;
        $this->b = $b;
    }
}

$a = new Foo();
$a->set(2, 'string');
echo serialize($a);
```

```
O:3:"Foo":2:{s:6:"?Foo?a";i:2;s:4:"?*?b";s:6:"string";}
```

sleep ו-wakeup (המשך...)

- תהליך ההיפוך של serialize לוקח את המחרוזות ומייצר ממנה את האובייקט.

```
$b = unserialize('O:3:"Foo":2:{s:6:"?Foo?a";i:2;s:4:"?*?b";s:6:"string";}');
var_dump($b);
```

```
object(Foo)#2 (2) {
  ["a:private"]=>
  int(2)
  ["b:protected"]=>
  string(6) "string"
}
```

- sleep היה הפונקציה שנקראת כאשר אנו מבצעים serialize על אובייקט. הפונקציה צריכה להחזיר מערך עם שמות השדות אותם אנו רוצים לשמור בתהליך הסריאליזציה.

```
public function __sleep() {
    return array('a');
}
```

sleep ו-wakeup (המשך...)

- wakeup הינה הפונקציה הנקראת בעת ביצוע unserialize על אובייקט. בפונקציה זו אנו אמורים לדאוג לשחזר את האובייקט למצב הקודם, כמו יצירת משאבים.

```
class Foo {

    private $filename;
    protected $handle;

    public function __construct($file) {
        $this->a = $file;
        $this->_openFile();
    }
    public function __wakeup() {
        $this->_openFile();
    }
    public function __sleep() {
        return array('filename');
    }
    private function _openFile() {
        $this->handle = fopen($this->filename, 'r');
    }
}
```

toString

- הפונקציה toString מאפשרת להחליט איך יתנהג האובייקט במקרה של המרה למחרוזת.

```
class Foo {
    protected $name = 'my name';
}
$a = new Foo();
echo $a;
```

```
Catchable fatal error:  Object of class Foo
 could not be converted to string
```

```
class Foo {
    protected $name = 'my name';

    public function __toString() {
        return 'Foo:' . $this->name;
    }
}
$a = new Foo();
echo $a;
```

```
Foo:my name
```

clone

- פונקציה זו נקראת בעת העתקת אובייקט על ידי שימוש ב-clone, והיא מאפשרת שליטה בתהליך ההעתקה, מעבר למה ש-php מבצעת (העתקת משתנים בלבד).
- בפונקציית clone נטפל בדרך כלל במשאבים, אובייקטים (העתק?) וכו..

```
class Foo {
    static public $counter = 0;

    public function __construct() {
        ++self::$counter;
    }

    public function __clone() {
        ++self::$counter;
    }
}
$a = new Foo();
$b = clone $a;

echo Foo::$counter;
```

```
2
```

setters & getters

- פונקציות אלו נועדו לספק גישה עקיפה לנתונים. דוגמה לאובייקט המשמש כאובייקט מכולה:

```
class Foo {

    private $container = array();

    public function __get($name) {
        if (array_key_exists($name, $this->container)) {
            return $this->container[$name];
        }
        return null;
    }

    public function __set($name, $value) {
        $this->container[$name] = $value;
    }
}

$a = new Foo();
$a->name = 'ziv';
echo $a->name;
```

```
ziv
```

call

- פונקציה זו דומה ל-get/set אלא שהיא נקראת עם קריאה לפונקציה.

```
class Foo {

    public function __call($name, $args) {
        echo "function Foo::$name, not exist";
    }
}

$a = new Foo();
$a->tryme();
```

```
function Foo::tryme, not exist
```

- דוגמה לשימוש בפונקציה הזו, הוא מודל ה"הכלה" המאפשר לנו ירושה מרובה...

call (המשך...)

- הכלה כירושה מרובה:

```
class A { //extending Foo & Bar

    private $foo;
    private $bar;

    public function __construct() {
        $this->foo = new Foo();
        $this->bar = new Bar();
    }

    public function __call($name, $args) {
        if (method_exists($this->foo, $name)) {
            return call_user_func_array(array($this->foo, $name), $args);
        }
        if (method_exists($this->bar, $name)) {
            return call_user_func_array(array($this->bar, $name), $args);
        }
        die('Fatal error: function not exist...');
    }

}
```

```
class Bar {

    public function bar() {}
}
```

```
class Foo {

    public function foo() {}
}
```

isset & unset

- כמו get ו-set, מאפשרים לנו isset ו-unset גישה לא ישירה למאפיינים של האובייקט.
- נרחיב את המכולה שהצגנו בדוגמת ה-setters/getters:

```
class Foo {

    private $container = array();

    public function __isset($name) {
        return array_key_exists($name, $this->container);
    }

    public function __unset($name) {
        unset($this->container[$name]);
    }
    ...
}
$a = new Foo();
var_dump(isset($a->name));
$a->name = 'ziv';
var_dump(isset($a->name));
unset($a->name);
var_dump(isset($a->name));
```

```
false
true
false
```

Design Patterns

תבניות עיצוב

"בהנדסת תוכנה, תבנית עיצוב היא פתרון כללי לבעיה שכיחה בעיצוב תוכנה. תבנית עיצוב אינה עיצוב סופי שניתן להעבירו הישר לקוד, אלא תיאור או תבנית לדרך לפתרון בעיה, שעשויים להיות שימושיים במצבים רבים. תבניות עיצוב מונחות עצמים מציגות לרוב יחסים וקשרי גומלין בין מחלקות או אובייקטים, בלי לפרט את מחלקות או אובייקטי היישום הסופיים המעורבים. אלגוריתמים אינם נחשבים כתבנית עיצוב, כיוון שהם פותרים בעיות חישוביות ולא בעיות עיצוב."

וויקיפדיה עברית

singleton

- תבנית זו נועדה על מנת לאפשר לנו ליצור מופע יחיד של אובייקט בכל האפלקציה שלנו.

```
class Foo {

    private static $instance = null;

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

    private function __construct(){}
}
```

```
$a = new Foo();
```

```
Fatal error:  Call to
 private Foo::__construct()
 from invalid context
```

singleton (המשך...)

- נבחן אם אכן נוצר רק מופע בודד...

```
$a = Foo::getInstance();
$b = Foo::getInstance();
$c = clone $a;
var_dump($a, $b, $c);
```

```
object(Foo)#1 (0) {
}
object(Foo)#1 (0) {
}
object(Foo)#2 (0) {
}
```

- הפלט מראה לנו ש a ו-b הם אכן אותו אובייקט, אך c הוא אובייקט נוסף. כדי לתקן את הבעיה הזו, נוסיף את הפונקציה clone כפרטית על מנת למנוע ביצוע clone.

```
private function __clone() {}
```

registry

- אחת הבעיות הנפוצות בתכנות, הינה שימוש במשתנים גלובליים (זיהום ה namespace). כאשר אנו זקוקים לגישה גלובלית לאובייקט, נשתמש בתבנית ה-registry שמשמת מכולה גלובלית לגישה לנתונים.

```
class Registry {

    private $container = array();
    public function __get($name) {...}
    public function __set($name, $value) {...}

    /** singleton **/
    private static $instance = null;

    public function getInstance() {
        if (is_null(self::$instance)) {
            self::$instance = new self();
        }
        return self::$instance;
    }
    private function __construct() {}
    private function __clone() {}
}
```

registry (המשך...)

- שימוש ב-registry:

```
$registry = Registry::getInstance();
$registry->obj = new MyObject();
```

```
Registry::getInstance()->obj->func();
```

```
$obj = Registry::getInstance()->obj;
$obj->func();
```

factory

- תבנית זו באה לפתור את בעית יצירת אובייקטים מורכבים במהלך התכנית. כאשר אנו צריכים לייצר אובייקט מורכב – אנו צריכים לדעת איזה פרמטרים הוא צריך לקבל וכו...
- תבנית זו פותרת לנו את הבעיה על ידי אספקת תשתית אחידה לייצור אובייקטים.

- בעיה: האפליקציה שלנו מסוגלת לעבוד עם מספר סוגים של מסדי נתונים. אני מעוניין לפתור את בעיית יצירת החיבורים למסד הנתונים כך שבכל מקום בקוד בו אזדקק ליצירת קישור שכזה, לא אזדקק לאסוף את כל הפרמטרים ולשנות את הקוד ביותר ממקום בודד.
- נדגים פתרון באמצעות factory.

factory (המשך...)

- לרשותינו המחלקות הבאות, המממשות את הממשק DbConnection:

```
interface DbConnection {

}
```

```
class MySql implements DbConnection {

}
```

```
class Oracle implements DbConnection {

}
```

```
class MSSql implements DbConnection {

}
```

```
class SqlLight implements DbConnection {

}
```

factory (המשך...)

- באמצעות factory, נוכל לשנות את הקריאה ליצירה בכל מקום בקוד – ולעדכן את הקבועים הלוונטים רק במקום בו הם מוגדרים.

```
class DbFactory {

    public static function getMySql() {
        return new MySql(HOST, USER, PASS);
    }

    public static function getOracle() {
        return new Oracle(HOST, DESC);
    }
    ...
}
```

- גישה אחרת, היא שימוש בפרמטרים...

factory (המשך...)

```
class DbFactory {

    const MYSQL    = 0;
    const ORACLE   = 1;
    const MSSQL    = 2;
    const SQLIGHT = 4;

    public static function getDb($type) {
        switch ($type) {
            case self::MYSQL:
                return new MySql(HOST, USER, PASS);
                break;

            case self::ORACLE:
                return new Oracle(HOST, DESC);
                break;
                        ...
        }
    }
}
```

```
$conn = DbFactory::getDb(Registry::getInstance()->db);
```

adapter

- תבנית זו נועדה להפוך לאפשר לאובייקט להציג את עצמו כאובייקט אחר, כלומר – לאפשר לאובייקט להתאים לממשק.

```
interface Shape {

    public function getArea();

    public function getPerimeter();
}
```

```
class Book {

    public function getTitle() {}

    public function getSize() {}

    public function getAuthor() {}
}
```

```
class BookShapeAdapter implements Shape {

    private $book;

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

    public function getArea() {
      return $this->book->getSize() *2;
    }

    public function getPerimeter() {
      return $this->book->getSize()/3;
    }

}
```

delegation

- בתבנית זו, אובייקט מתנהג למראית עין כמו אובייקט אחר.

```
class Foo {

    public function a() {}

    public function b() {}
}
```

```
class Bar {

    private $foo;

    public function __construct() {
        $this->foo = new Foo();
    }

    public function a() {
        return $this->foo->a();
    }

    public function b() {
        return $this->foo->b();
    }
}
```

delegation (המשך...)

- כבר ראינו דוגמה לשימוש ב-delegation כאשר ביצענו הורשה מרובה על ידי הכלה.

```
class A { //extending Foo & Bar

    private $foo;
    private $bar;

    public function __construct() {
        $this->foo = new Foo();
        $this->bar = new Bar();
    }

    public function __call($name, $args) {
        if (method_exists($this->foo, $name)) {
            return call_user_func_array(array($this->foo, $name), $args);
        }
        if (method_exists($this->bar, $name)) {
            return call_user_func_array(array($this->bar, $name), $args);
        }
        die('Fatal error: function not exist...');
    }

}
```

```
class Bar {

    public function bar() {}
}
```

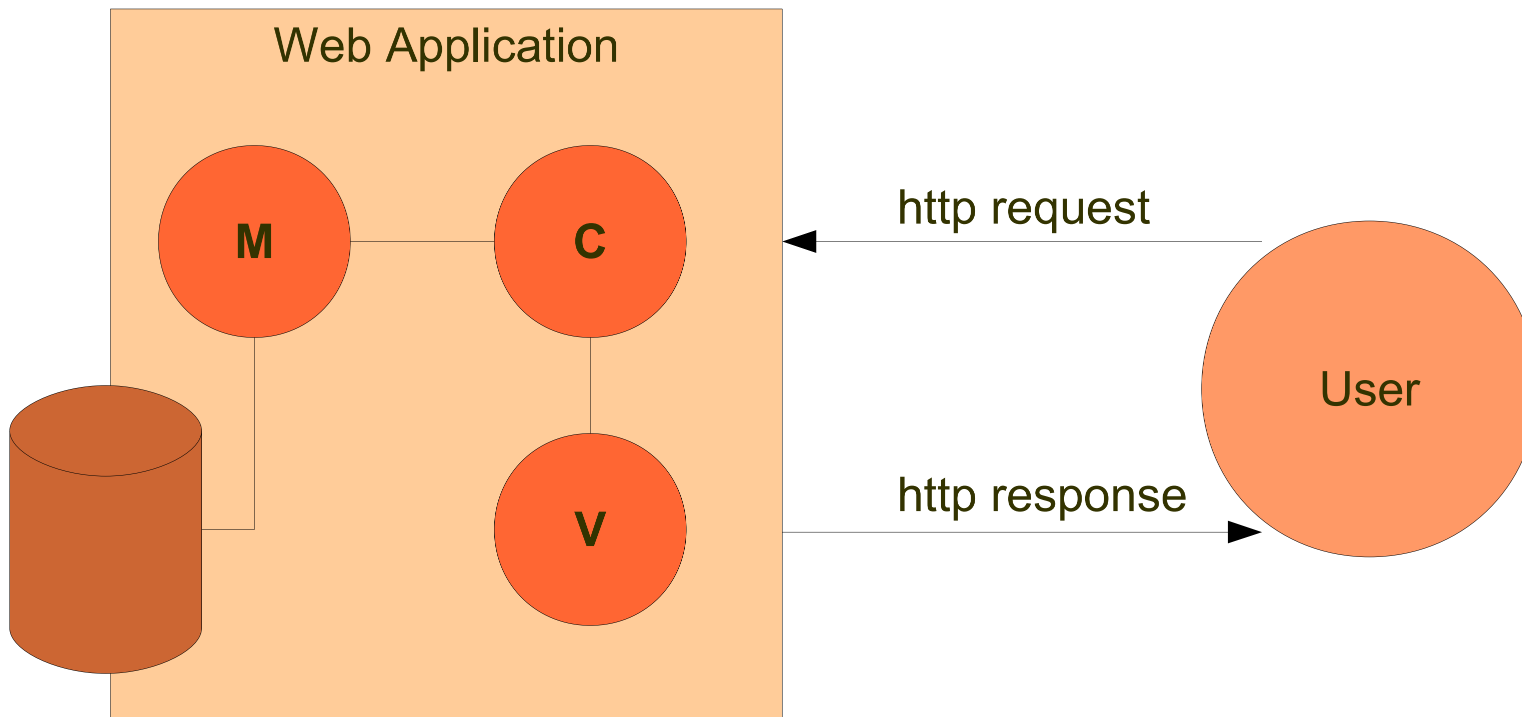
```
class Foo {

    public function foo() {}
}
```

MVC

- **Model View Controller Pattern הינה תבנית פיתוח לתיאור תהליכים בתכנית – או תבנית ארכיטקטונית. מאחר והתבנית מספקת תיאור כללי ביותר – אין דרך מימוש יחידה או מועדפת – אלא מספר גישות לשימוש בה תוך יישום העקרונות הבסיסיים.**
- **בדרך כלל מתוארים הרכיבים כך:**
  - Model – היחידה האחראית על ה-business logic, כלומר המידע, האינטרקציה עם המידע והמניפולציה של המידע של התכנית.
  - View – החלק המודל האחראי על התצוגה.
  - Controller – הינו החלק האחראי על הלוגיקה של התכנית.

MVC (המשך...)

- נהוג להשתמש בארכיטקטורה זו למימוש אפליקציות ברשת האינטרנט מאחר והיא מתאימה לגישת השרת/לקוח כמו שהיא מיושמת ב-web.



MVC (המשך...)

- דוגמה למימוש בסיסי של mvc (הדגמה)

תודה

The PHP Company