Le Design Pattern en PHP - Builder

Le ***monteur*** (***builder)*** est un modèle de conception souvent mal compris, confondu avec d'autres patterns de type construction.

L'objectif du monteur est de séparer le processus de construction de l'objet de sa représentation finale. En d'autres termes, cela signifie que le processus de construction est identique mais que le produit finit peut varier.

Un cas d'usage simple

Supposez que vous devez, à partir d'un logiciel de comptabilité, extraire vos dépenses mensuelles. Vous souhaitez pouvoir obtenir un rapport de dépenses sous la forme d'un tableau et aussi sous la forme d'un graphe.

Le processus de création sera probablement identique lorsqu'il va s'agir de construire le rapport (extraire les données de la base, ajouter des dépenses au rapport), mais le produit final ne sera pas du tout le même (une image contre un fichier au format HTML).

Le mauvais réflexe

Le mauvais réflexe consiste à développer plusieurs méthodes « en dur » qui exploiteront les données : Une méthode qui va générer du HTML, l'autre méthode qui va générer une image. Ces deux méthodes exploiteront elles même les données et dupliqueront les efforts.

Par exemple pour la version tabeau HTML:

```
$htmlOutput = '<table>' ;
foreach ($datas as $data){
    $htmlOutput .=
"<tr><td>".$data['Type']."</td><td>".$data['value']."</td></tr>" ;
}
$htmlOutput.="</table>"
```

Et pour la version en graphe

```
$graph = new Graph();
foreach ($datas as $data){
   $graph->addSlice($data['Type'], $data['value']);
}
```

Cette solution présente plusieurs inconvénients :

- Le code en charge d'exploiter les données est dupliqué (ici le parcours du tableau)
- Le code en charge de convertir les données (transformation HTML / Image) et celui en charge de les lire (parcours et lecture des lignes) sont mélangés
- Le code permettant de générer un rapport dans un format particulier n'est pas isolé, donc peu réutilisable à partir d'autres données

La solution du monteur / builder

Nous avons dit que le monteur permettait d'isoler le processus de création de la représentation finale du produit crée.

Commençons par identifier ce dont est constitué un rapport

- Un titre
- Une légende
- Des dépenses (libellé / valeur)

Nous pouvons ainsi isoler les étapes de création d'un rapport et définir nos monteurs :

```
interface IReportBuilder
{
    public function addTitle($pTitle);
    public function addLegend($pLegend);
    public function addExpense($pType, $pAmount);
}
```

Maintenant que nous disposons de cette interface de création, nous pouvons mettre à jour notre méthode de génération de rapport (appelée directeur) afin qu'elle utilise l'interface du monteur.

```
class ReportDirector
{
    public function createExpenseReport(IReportBuilder $pReportBuilder)
    {
        $data = $this->getExpenseData();
        $pReportBuilder->addTitle($data['meta']['title']);
        $pReportBuilder->addLegend($data['meta']['legend']);

        foreach ($data['datas'] as $line){
            $pReportBuilder->addExpense($line['type'], $line['amount']);
        }
        return $pReportBuilder;
    }
//... code de récupération des données de dépense
}
```

Le constat est simple : le code qui génère le rapport se moque de savoir s'il doit générer du HTML, une image ou un fichier CSV, ce qui l'importe est d'*assembler* le produit fini en le définissant pièce par pièce.

Développons maintenant nos monteurs

Le monteur de type HTML

Le monteur de type HTML va s'assurer de la bonne prise en charge des caractères spéciaux et générer un tableau HTML.

```
class HTMLReportBuilder implements IReportBuilder
{
    //... protected attributes
    public function addTitle($pTitle)
    {
        $this->_title = htmlentities($pTitle);
    }

    public function addLegend($pLegend)
    {
        $this->_legend = htmlentities($pLegend);
    }

    public function addExpense($pType, $pAmount)
    {
        $this->_expenses .= '<tr><td>'.
                    htmlentities($pType).'</td><td>'.
                    $pAmount.'</td></tr>';
    }

    public function getReport()
    {
       return '<h2>'.$this->_title.'</h2>'
               .'<p>'.$this->_legend.'</p>'
               .'<table>'
               .'<tr><th>Type de dépense</th><th>Montant</th>'
               .$this->_expenses
               .'</table>';
    }
}
```

Le monteur de type Image

Le monteur de type Image va générer une image aux dimensions paramétrables avec des sections à la taille proportionnelle au montant de dépense.

```
class BarReportBuilder implements IReportBuilder
{
   //... protected attributes
   public function __construct ($pWidth, $pHeight, $pFilePath)
    {
        $this->_width = $pWidth;
        $this->_height = $pHeight;
        $this->_filePath = $pFilePath;
    }

    public function addTitle($pTitle)
    {
        $this->_title = $pTitle;
    }

    public function addLegend($pLegend)
    {
        $this->_legend = $pLegend;
    }

    public function addExpense($pType, $pAmount)
    {
```

```
        $this->_expenses[$pType] = $pAmount;
        $this->_max += $pAmount;
    }

    public function getReport()
    {
        //creation de l'image
        $image = imagecreate($this->_width, $this->_height);

        //black filler
        $black = imagecolorallocate($image, 255, 255, 255);
        imagefilledrectangle($image, 0, 0 , $this->_width, $this->_height,
$black);

        //now we're gonna fill the datas
        $indice = 0;
        $xPosition = self::BORDER_WIDTH;
        foreach ($this->_expenses as $type=>$expense) {
            $color = $this->_getColor($indice, $image);
            imagefilledrectangle($image,
                                 $xPosition,
                                 self::BORDER_WIDTH,
                                 $xPosition+($movedBy = ($expense/$this-
>_max) * ($this->_width-self::BORDER_WIDTH*2)),
                                 $this->_height-self::BORDER_WIDTH,
                                 $color
                                );
            $xPosition += $movedBy;
            $indice++;
        }
        imagegif($image, $this->_filePath);
    }
}
```

Code Example

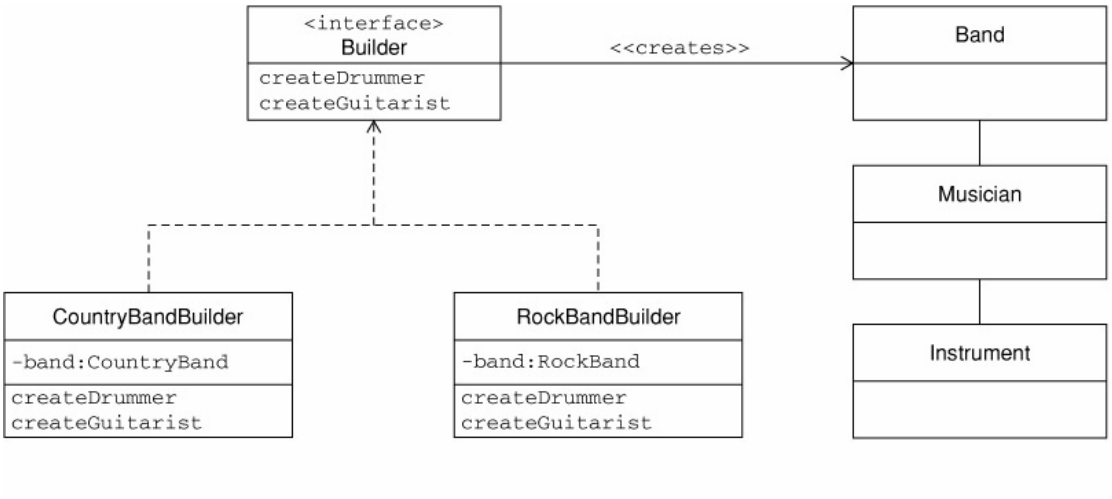
```
<?php
require 'builders/ireportbuilder.php';
require 'builders/htmlreportbuilder.php';
require 'builders/clireportbuilder.php';
require 'builders/barreportbuilder.php';
require 'reportdirector.php';

/*
 *  creaza formatul pentru tipul de raport dorit
 *  - grafic => obiect de tipul IReportBuilder realizat cu ajutorul clasei
BarRaportBuilder
 *  - html   => obiect de tipul IRaportBuilder realizat cu ajutorul clasei
HTMLRaportBuilder
 *  - text   => obiect de tipul IRaportBuilder realizat cu ajutorul clasei
CliRaportBuilder
 */
 //$builder = new BarReportBuilder(300, 40, '../builder/tmp/image.gif');
 $builder = new HTMLReportBuilder();
 //$builder = new CliReportBuilder();

/*
 * creeaza obiect de tip ReportDirector
 */
 $director = new ReportDirector();

 //$tst = $director->createExpenseReport($builder)
 /*
  * afiseaza tipul de raport pentru obiectul creat
  *
  */
$director->displayInfo($director->createExpenseReport($builder));
?>


class ReportDirector
{

       /*
       *  metoda pentru creerea tipul de raport dorit in care se paseaza un
       *  obiect de tipul IReportBuilder ( clase care implementeaza interfata
IReportBuilder )
       *
       *  clase ce implementeaza IReportBuilder
       *   => BarReportBuilder  ( format grafic )
       *   => CliReportBuilder  ( format text   )
       *   => HTMLReportBuilder ( format HTML   )
       */
   public function createExpenseReport(IReportBuilder $pReportBuilder) {

             // incarca datele din metoda statica getExpenseData()
             $data = ReportDirector::getExpenseData();

             // creaza titlul pentru obiectul de tip IReportBuilder
             $pReportBuilder->addTitle($data['meta']['title']);

             // create legenda pentru obiectul de tip IReportBuilder
        $pReportBuilder->addLegend($data['meta']['legend']);

        // incarca liniile de date pentru obiectul de tip IReportBuilder
        foreach ($data['datas'] as $line){
            $pReportBuilder->addExpense($line['type'], $line['amount']);
        }

        return $pReportBuilder;
    }
```

```
    /*
     *  metoda pentru stocarea datelor ( echivalenta unei baze de date )
     */
    protected static function getExpenseData()
    {

        return array('meta'=>array('title'=>'Déepenses mensuelles',
'legend'=>'Mois de Janvier 2011'),
                     'datas'=>array(
                                    array('type'=>'Livres', 'amount'=>157),
                                    array('type'=>'Matériel',
'amount'=>200),
                                    array('type'=>'Services',
'amount'=>377),
                                    array('type'=>'Boissons', 'amount'=>50)
                                   )
                    );
    }

    /*
     *  metoda pentru afisarea datelor in functie de tipul de raport dorit
     *  obiectul primit ca parametru este un obiect de tipul IReportBuilder
     */
    public function displayInfo(IReportBuilder $pReportBuilder){

       echo $pReportBuilder->getReport();
    }

}

interface IReportBuilder {

    public function addTitle($pTitle);

    public function addLegend($pLegend);

    public function addExpense($pType, $pAmount);
}
//end interface

class BarReportBuilder implements IReportBuilder
{
    protected $_title;
    protected $_legend;
    protected $_expenses = array();
    protected $_max = 0;

    protected $_width;
    protected $_height;

    const BORDER_WIDTH = 4;

    public function __construct ($pWidth, $pHeight, $pFilePath)
    {
        $this->_width = $pWidth;
        $this->_height = $pHeight;
        $this->_filePath = $pFilePath;
    }

    public function addTitle($pTitle)
    {
        $this->_title = $pTitle;
    }

    public function addLegend($pLegend)
    {
        $this->_legend = $pLegend;
```

```
    }

    public function addExpense($pType, $pAmount)
    {
        $this->_expenses[$pType] = $pAmount;
        $this->_max += $pAmount;
    }

    public function getReport()
    {
        //creation de l'image
        $image = imagecreate($this->_width, $this->_height);

        //black filler
        $black = imagecolorallocate($image, 255, 255, 255);
        imagefilledrectangle($image, 0, 0 , $this->_width, $this->_height,
$black);

        //now we're gonna fill the datas
        $indice = 0;
        $xPosition = self::BORDER_WIDTH;
        foreach ($this->_expenses as $type=>$expense) {
            $color = $this->_getColor($indice, $image);
            imagefilledrectangle($image,
                                 $xPosition,
                                 self::BORDER_WIDTH,
                                 $xPosition+($movedBy = ($expense/$this-
>_max) * ($this->_width-self::BORDER_WIDTH*2)),
                                 $this->_height-self::BORDER_WIDTH,
                                 $color
                                );
            $xPosition += $movedBy;
            $indice++;
        }
        //imagegif($image, $this->_filePath);

        // affiche l'image sur le navigateur
             header('Content-Type: image/gif');
             imagegif($image);
    }

    private function _getColor($pIndice, $pForImage)
    {
        $max = count($this->_expenses)+1;
        $color = imagecolorallocate($pForImage,
                                    (int) (($pIndice+(($pIndice%3 === 0) ? 1
: 0)) / $max * 256) ,
                                    (int) (($pIndice+(($pIndice%3 === 1) ? 2
: 0)) / $max * 256),
                                    (int) (($pIndice+(($pIndice%3 === 2) ? 3
: 0)) / $max * 256));
        return $color;
    }
}

?>

class CliReportBuilder implements IReportBuilder
{
    private $_title;
    private $_legend;
    private $_expenses;

    public function addTitle($pTitle)
    {
        $this->_title = $pTitle;
    }
```

```
    public function addLegend($pLegend)
    {
        $this->_legend = $pLegend;
    }

    public function addExpense($pType, $pAmount)
    {
        $this->_expenses .= $pType.' : '.$pAmount."\n\r";
    }

    public function getReport()
    {
       return $this->_title.' ('.$this->_legend.")\n\r"
               .$this->_expenses;
    }

}

/*
 *  clasa HTMLRaportBuilder implementeaza interfata IReportBuilder
 *  si creaza raportul de tip HTML
 */
class HTMLReportBuilder implements IReportBuilder
{
    private $_title;
    private $_legend;
    private $_expenses;

    public function addTitle($pTitle)
    {
        $this->_title = htmlentities($pTitle);
    }

    public function addLegend($pLegend)
    {
        $this->_legend = htmlentities($pLegend);
    }

    public function addExpense($pType, $pAmount)
    {
        $this->_expenses .=
'<tr><td>'.htmlentities($pType).'</td><td>'.$pAmount.'</td></tr>';
    }

    public function getReport()
    {
       return '<h1>'.$this->_title.'</h1>'
               .'<p>'.$this->_legend.'</p>'
               .'<table bgcolor="#CCCCCC" width="50%" cellspacing="1"
cellpadding="2" style="border:1px solid #000000">'
               .'<tr><td>Type de dépense</td><td>Montant</td>'
               .$this->_expenses
               .'</table>';
    }

}
```

The Builder Pattern



If you look back at the composite pattern, you'll notice that the objects that made up the composite were manually created, as shown in the following example:

```
$drums = new DrumSet("tama maple set");
$drums->add(new SnareDrum("snare drum"));
$drums->add(new BaseDrum("large bass drum"));

$cymbals = new Cymbal("zildjian cymbal set");
$cymbals->add(new Cymbal("small crash"));
$cymbals->add(new Cymbal("large high hat"));
$drums->add($cymbals);
```

The code creating this DrumSet composite is not particularly complex, but what if you needed to create Instrument or Instrument composite objects for many different Musician objects, and those Musician objects in turn were composites of a Band object?

It wouldn't be impossible to write out the code that created the new objects and then added them, although it might start to get tedious if you had several bands with several musicians in them, each with their own set of instruments. If you needed to create bands in response to the actions of a user, perhaps through a GUI, then it might start to get difficult—especially if you had to hard code each type of band, with the required musicians and instruments.

Implementation

Say that you were required to create a wizard that generated a Band for an end user. The user would choose from a list of genres (for example, rock, country, salsa, heavy metal, and so on), and the resulting Band would be returned, assembled with its musicians and their instruments. Figure 4-13 shows the class diagram for the builder pattern

For this example, the Builder pattern creates Bands that have only two types of Musicians, guitarists and drummers.

```
<?php

interface Builder {
    public function buildDrummer();
    public function buildGuitarist();
}
?>
```

Each time a concrete builder class is instantiated, it creates a Band object. Take a look at the code for the RockBandBuilder class that follows.

```
<?php

require_once("interface_builder.php");
require_once("class_rockband.php");
require_once("class_musician.php");
require_once("class_instrument.php");

class RockBandBuilder implements Builder {

  private $band;

  function __construct($name) {
        $this->band = new RockBand($name);
  }

  public function getBand() {
        return $this->band;
  }

  public function buildDrummer() {

        $musician = new Musician("rock drummer");

        $drumset = new Instrument("rock drum kit");
        $drumset->add(new Instrument("cymbal"));
        $drumset->add(new Instrument("bass drum"));
        $drumset->add(new Instrument("snare drum"));

        $musician->addInstrument($drumset);
        $this->band->addMusician($musician);
  }

  public function buildGuitarist() {

        $musician = new Musician("rock guitarist");

        $guitar = new Instrument("electric guitar");

        $musician->addInstrument($guitar);
        $this->band->addMusician($musician);
  }

}
?>
```

See how the implementation of the constructor in RockBandBuilder creates a RockBand? Similarly, the constructor of CountryBandBuilder creates a CountryBand, shown next, in the file countryband_builder.php.

```
<?php
require_once("interface_builder.php");
require_once("class_musician.php");
require_once("class_countryband.php");
require_once("class_instrument.php");

class CountryBandBuilder implements Builder {
  rivate $band;
```

```
    function __construct() {
        $this->band = new CountryBand();
    }

    public function getBand() {
        return $this->band;
    }

    public function buildDrummer() {

        $musician = new Musician("washboard player");

        $drumset = new Instrument("washboard");

        $musician->addInstrument($drumset);
        $this->band->addMusician($musician);
    }

    public function buildGuitarist() {

        $musician = new Musician("country guitarist");

        $guitar = new Instrument("acoustic guitar");

        $musician->addInstrument($guitar);
        $this->band->addMusician($musician);
    }
}
?>
```

Check out the buildDrummer() method from RockBandBuilder. Now compare it to the one from CountryBandBuilder. Notice how each method creates not only the Musician object but also the instruments for that musician. Because of the Builder interface, both Builders are required to implement the buildDrummer() and buildGuitarist() methods, but they each construct a Musician in a completely different way, including the Instrument objects belonging to that musician.

You may have noticed that the Musician and Instrument classes are no longer subclassed into specific subtypes such as Guitarist. We did this to demonstrate that Builders can be implemented to create different objects—as in the constructor—or the same objects with different parameters and operations, as in the buildDrummer() method.

The Director

Builder patterns have one final important aspect, called the Director. The Director is responsible for calling the methods of the Builder to create the finished product, which in this case is a Band object. In this example, the Director will be the Application object similar to the one described in the facade pattern. The Application class follows in a file called application.php:

```
<?php

class Application {
  public static function createBand(Builder $builder) {

        $builder->buildGuitarist();
        $builder->buildDrummer();
```

```
        return $builder->getBand();
    }
}
?>
```

The Application class has one method, createBand, which takes a Builder as its argument. The Application then calls the create methods of the passed Builder object. You should note two important points here. First, the Application decides which methods of the Builder it wants to call; second, it doesn't care which type of Builder is passed to it.

Finally, here is how you would use the Application (a.k.a. the Director) and Builder together. Simply create an instance of the appropriate Builder object and pass it to the createBand method of the Application.

```
$builder = new RockBandBuilder();
$band = Application::createBand($builder);
```

The Application object and Builder work together to build the correct Band object step by step. The Band object is manipulated and stored in the Builder until it's requested by the getBand method.

Considerations

The builder pattern is useful for assembling composites and hierarchical object structures like the Band->Musician->Instrument one. Although having the Builder return a completed object is useful, don't forget that the Band object returned by the Builder is still the same as any Band object you might have created by hand. That means that it can still be altered after its construction.

The Director class can be modified to allow for more than one creation method. In the case of the Application object, you had only one method, createBand, but having methods that called different configurations using the same builder is possible. For example, you could have methods such as createTrio. By passing different Builder objects, the same method would assemble either a rock trio or a country trio.